commodore 64 & 128

COMMODORE'S HANDBOOK OF

SIMONS' BASIC

commodore COMPUTERS

JANE G. REH
Foreword by David Simons

Brady

REH

COMMODORE'S HANDBOOK OF SIMONS' BASIC

Brady

Commodore's Handbook of Simons' BASIC

Jane G. Reh

Brady Communications Company, Inc.
A Prentice-Hall Publishing Company
Bowie, MD 20715

Commodore's Handbook of Simons' BASIC

Copyright © 1985 by Brady Communications Company, Inc.
All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage and retrieval system, without permission in writing from the publisher. For information, address Brady Communications Company, Inc., Bowie, Maryland 20715.

Library of Congress Cataloging in Publication Data
Reh, Jane G.
Commodore's Handbook of Simons' BASIC.

Includes index.
1. Commodore 64 (Computer)—Programming.
2. Basic (Computer program language) I. Title.
QA76.8.C64R44 1985 001.64'24 85-5903

ISBN 0-89303-419-3

Prentice-Hall of Australia, Pty., Ltd., *Sydney*
Prentice-Hall Canada, Inc., Scarborough, *Ontario*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall Do Brasil LTDA., *Rio de Janeiro*
Whitehall Books, Limited, Petone, *New Zealand*

Printed in the United States of America

85 86 87 88 89 90 91 92 93 94 95 1 2 3 4 5 6 7 8 9 10

Publishing Director: David Culverwell
Acquisitions Editor: Gisele Asher
Production Editor/Text Designer: Roberta Glencer
Art Director/Cover Design: Don Sellers
Assistant Art Director: Bernard Vervin
Manufacturing Director: John Komsa

Typesetter: Shepard & Poorman, Indianapolis, IN
Printer: R. R. Donnelley & Sons, Harrisonburg, VA
Typefaces: Mallard (text & display), OCR-B (programs)

Contents

Foreword vii

Preface ix

I Programming Aids We All Need

Introduction 1

1 Statement Numbers and Program Listings 4

Automatic Line Numbers (AUTO) 4
Renumbering Statements (RENUMBER) 6
Listing Programs in Pages (PAGE) 9
Controlling Listing Speed (DELAY and the SHIFT key) 11
Reversing the NEW Command (OLD) 12
Cold Start (COLD) 12

2 Debugging Aids 14

Program Walk-Through (TRACE) 14
Recheck the Program (RETRACE) 17
Checking the Value of Variables (DUMP) 18
Searching for Codes and Character Strings (FIND) 19
Temporary Program Hold (PAUSE) 20
Combining Program Aids (TRACE, DUMP, PAUSE) 20
Control your "Crash"! (ON ERROR and NO ERROR) 21

3 Program Security Aids 26

Hiding your Code (DISAPA) 26
Securing your Code (SECURE) 27

4 Assigning Commands to the Function Keys 31

Making Assignments (KEY) 31
Listing Assignment Commands (DISPLAY) 34

5 Added Math Functions 36

Returning the Quotient (DIV) 36
Returning the Remainder (MOD) 37
Returning the Fractional Part (FRAC) 37
Binary Number Conversion (%) 38
Hexadecimal Number Conversion ($) 38
Bit to Bit—the Exclusive OR (EXOR) 38

6 Disk Handling 40

Simplified Disk Entries (DISK) 40
Directory Assistance (DIR) 41

II Added INPUT and OUTPUT Options

Introduction 43

7 Character String Input Options 46

Centering Strings (CENTRE) 46
Inserting Characters (INSERT) 48
Changing Characters (INST) 49
Character Search (PLACE) 51
Duplicating Strings (DUP) 52
Switching Between Character Sets (CSETØ and CSET1) 54
Reversing Character Background (INV) 55

8 Input Validation 57

Branching Using Function Key Input (INKEY) **57**
Branching Using Keyboard Input (ON KEY) **59**
Branching Back to ON KEY (RESUME) **61**
Controlling and Limiting Keyboard Input (FETCH) **62**

9 Output Options 68

Positioning Text on the Screen (AT) **68**
Formatting Numbers on the Screen (USE) **70**
Duplicating a Screen Section (MOVE) **71**
Highlight a Listing (OPTION 1Ø and OPTION Ø) **74**
Printing Hard Copy (HRDCPY) **75**

III More Programming Power

Introduction *77*

10 Condition Testing and Controlling Loops 79

ELSE at Last! (IF . . . THEN . . . ELSE) **80**
Reusing Conditions (RCOMP ELSE) **81**
Testing at the End of a Loop (REPEAT UNTIL) **82**
Loop with Multiple Exit Testing (LOOP, EXIT IF, END LOOP) **83**
Computed GOTO (CGOTO) **85**

11 Data Handling 89

Moving Data Pointers (RESET) **89**
Reusing Variables (LOCAL and GLOBAL) **93**

12 Structured Programming 96

Naming a Procedure (PROC) **97**
Calling an Open-Ended Procedure (CALL) **97**
Calling a Closed Procedure (EXEC) **99**
Ending a Closed Procedure (END PROC) **99**
Building Programs Through Mergers (MERGE) **103**

IV Low-Resolution Color and Graphics

Introduction **105**

13 Color and Characters 107

Border and Screen Color (COLOUR) **107**
Character Background Color (BCKGNDS) **108**
Flashing Colorful Characters (FLASH) **112**
Flashing the Screen Border (BFLASH) **115**
Colorful Character Displays (FILL, FCOL, FCHR) **116**
Saving and Restoring a Screen (SCRSV and SCRLD) **120**

14 User-Defined Graphics 122

Replacing Commodore Characters (MEM and DESIGN 2) **122**
Creating a New Design (@) **123**
Scrolling the Screen for Character Movement (RIGHT, LEFT, UP and DOWN) **127**

V High-Resolution Graphics

Introduction **133**

Graphic Screens **133**

High-Resolution Screen **134**

15 Plotting Graphics 137

Initializing the High-Resolution Graphics Screen (HIRES) **137**
Plotting Rectangles (REC) **137**
Plotting Squares (REC) **140**
Adding Characters to a Graphics Screen (TEXT and CHAR) **142**
Changing Plotting Colors (LOW COL) **148**
Reverting to Original Plotting Colors (HI COL) **149**
Drawing Lines (LINE) **150**
Plotting Circles (CIRCLE) **152**
Drawing an Arc from a Circle (ARC) **155**
Plotting the Radius of a Circle (ANGL) **159**
Bit-by-Bit Plotting (PLOT) **161**
Checking Bit Status (TEST) **162**
Leaving a Graphics Screen and Returning to Low-Resolution (NRM) **164**

16 Coloring and Drawing in HIRES 165

Filling Shapes With Color (PAINT) **165**
Plottting a Color-Filled Rectangle or Square (BLOCK) **167**
Designing Shapes (DRAW) **168**
Rotating and Controlling the Size of Shapes (ROT) **170**

17 Graphic Output 174

Recalling a Graphics Screen Display (CSET 2) **174**
Printing Hardcopy (COPY) **175**

VI Multicolor Graphics

***Introduction* 177**

***Multicolor Screen Configuration* 177**

18 Multicolor Plotting 180

Initializing and Plotting in Multicolor (MULTI) **181**
Squares on a Multicolor Screen (REC) **183**
Circles, Angles, and Arcs (CIRCLE, ANGL, and ARC) **184**
Changing Colors (LOW COL) **185**
Painting Shapes (PAINT) **185**

VII Sprite Graphics

***Introduction* 187**

19 Creating and Using Sprites 189

Reserving Memory for a Sprite (DESIGN) **189**
Designing a Sprite (@) **191**
Initializing a Sprite (MOB SET) **194**
Added Color for a Multicolor MOB (CMOB) **195**
Displaying a Sprite (MMOB) **196**
Moving an On-Screen Sprite (RLOCMOB) **201**
Sprite Collisions (DETECT and CHECK) **201**
Sprite Eliminations! (MOB OFF) **203**

20 Sprites on a High-Resolution or Multicolor Screen **207**
Memory Address Changes (DESIGN and MOB SET) **207**
Program Statement Order when Displaying Both High-Resolution and Multicolor Sprites **208**
Another Programming Option (GRAPHICS) **210**
VIII Simons' Sound
Introduction **211**
Basic Music Facts **212**
21 Programming Music and Sound **214**
Preview of Commands **214**
How Loud—How Soft (VOL) **215**
Introduction to Waveforms **217**
Sound Waves (WAVE) **218**
SOUND with the Pulse Waveform (SOUND and POKE) **220**
Sound Shapes (ENVELOPE) **222**
Composing Music (MUSIC) **224**
Setting the Sound Loose (PLAY) **226**
Advanced Techniques **228**
IX Leaving the Keyboard
Introduction **233**
22 Joystick, Paddle, and Light Pen Read Functions **234**
Joystick Position and Firebutton Action (JOY) **234**
Paddle Knob Rotation Positions (POT Ø and POT 1) **237**
Light Pen Position Reading (PENX and PENY) **238**
Appendix A Simons' BASIC Command Reference Listing **239**
Low-Resolution Mode Commands *239*
Mathematic Functions *247*
Graphic Mode Commands *247*
Sprite Commands *251*
Music Synthesis Commands *252*
Video Game Controller Functions *253*
Appendix B List of Reference Tables and Screen Grids **255**
Index **257**

Foreword

For the home computer enthusiast, books are a very necessary and valuable complement to the hardware and software. In this book, Jane Reh has produced a comprehensive survey of Simons' BASIC that will be of enormous value to anyone who is exploring its many commands.

My first encounter with Commodore computers was with the early 2000 series PET's for which I started writing utilities, largely for my own use. When I was working with a Commodore Vic 20, I devised a selection of routines to enable me to write my programs more easily.

Then the 64 came along with its marvelous potential for graphics and sound! It was a logical step to write similar routines for the new machine. However, as I did more work on the 64, I had more and more ideas for commands that could be valuable; the section dealing with Sprites is a good example.

Gradually, Simons' BASIC came into being. With support and encouragement from Gail Wellington of Commodore (United Kingdom), I extended Simons' BASIC to produce the version now available.

There is a wealth of good software, of all types, available for most computers in the market. Sadly, much of this software is never used to its full potential by its purchasers. Many people are working alone at home on their computer, and I know from my own experience how frustrating it can be if no one explains exactly how to use some new program. Over the last year I have often had the pleasure of talking to Simons' BASIC users at exhibitions, and sometimes they have only been able to use half the capabilities because they felt unable to understand some of the other commands. Often, if time is short, a new user may feel too discouraged to continue, but a work such as Jane's with its abundance of examples and clear explanations can provide just the help that is needed.

I like to think that Simons' BASIC helps people to get more enjoyment from programming their 64; I am grateful to Jane for writing this book which will make such a contribution to that enjoyment.

David Simons
England, 1985

About The Author

Ms. Reh had an extensive career as a technical writer. She wrote and edited technical copy for technical magazines and prepared technical promotional material, reports, and proposals for several major corporations. While preparing system and user documentation for a software development company, Ms. Reh changed career paths.

As a programmer/analyst, data analyst, systems representative, and systems analyst, Ms. Reh developed a strong and diversified computer background. She assisted in the establishment of design criteria, testing criteria, and applications packages working with FORTRAN IV, INFOL, AND ALC, and assisted clients with the training of technical personnel in the use of proprietary systems and the INFOL language. Ms. Reh is experienced in data processing applications with particular emphasis in the design of information systems and the reduction and analysis of research data.

Ms. Reh now combines her dual careers as a full-time author, writing computer books and contributing articles to leading computer magazines.

Acknowledgments

Many Commodore personnel supported the publication of this book. A great deal of thanks goes to Jim Gracely and Diane LeBold. Jim Gracely also provided much technical assistance and suggestions. As always, I am grateful to all my reviewers, many of whose suggestions are a part of this book.

I am delighted to have a Foreword by David Simons, the brilliant author of the software about which this book is written. Thank you, David, for your support and technical assistance. It is a very important contribution to this book.

Of course this book wouldn't have made it to print if it weren't for the support of Brady Communications Company personnel. A large debt of gratitude goes to Gisele Asher, my Acquisitions Editor, for her deep and sustained support of this writing project. The excellent layout and editing of this book belong to my Book Production Editor, Roberta Glencer. Thank you, Bobbie.

Preface

Simons' BASIC is for everyone who programs a Commodore 64 or Commodore 128 computer. Students and businessmen alike will want to use the plotting graphics for creating bar charts, pie charts, and line graphs as an important addition to their reports. Game programmers and teachers and students writing educational programs will welcome the simplified use of graphics and sound. All Commodore 64 and 128 owners will welcome the versatility of this language and its programming ease and power. Simons' BASIC enables you to write programs creating beautifully and easily formatted output—whether numbers, text, graphics, or a combination of the three. This is the language for creative programmers.

Perhaps you have already bought the Simons' BASIC cartridge and are looking for help in learning to use it well. This reference book—which will replace your *Simons' BASIC User Guide*—will show you how to make the most of each command, by supplying all the DOs and DON'Ts. The User Guide supplied with the cartridge contains a number of command description and program errors, and leaves a great deal of the work of figuring out Simons' BASIC to the user. As you know, user guides are meant only as an introduction to the software, not as a definitive reference text or a tutorial, as this book is.

Perhaps you are simply curious about this extended BASIC for your Commodore. If you are considering buying it, reading this book will convince you that Simons' BASIC is a "must buy"! Your enjoyment in programming the Commodore will increase by being able to use more of the potential of your computer, taking advantage of over 100 additional programming commands.

When using Simons' BASIC, the resident BASIC of the Commodore 64 or Commodore 128 in the 64 mode remains unchanged. You can insert the cartridge and continue using BASIC as you have been, never using a Simons' BASIC command. Of course, this would be a sad loss of programming enjoyment. But it does enable you to use Simons' BASIC at your own pace, slowly becoming familiar with the new programming commands a few at a time. While all of your current BASIC programs will run on Simons' BASIC, your new programs using this extended BASIC language will require that the Simons' BASIC cartridge be active. For this reason, it will be important to note the language used in each program you store on tape or disk after becoming a Simons' BASIC programmer.

Simons' BASIC is a **large extension** of Commodore BASIC. The commands cover many different areas of programming. Therefore, for ease of use and reference, this book is divided into nine topic sections, each subdivided into chapters. Each chapter covers a single topic of closely related commands.

Since it's not possible to give a brief description of Simons' BASIC and do it justice, the following are summaries of each of the nine topic sections.

Section I: Programming Aids We All Need

These are the handy, time-saving commands that all programmers love! Automatic line numbering and renumbering, listing controls, and debugging aids make programming less tedious and more pleasurable. OLD will restore a

program that used to be permanently lost after executing NEW. There are program security aids for hiding code. Being able to define *16 functions* and use them with the flick of a function key makes inputing often-used commands easy. Think of the keyboarding time you will save (not to mention those typos)! In addition, there are six new numeric functions and two commands for simplified disk handling.

Section II: Added Input and Output Options

There are eight additional character input options, many of which simplify character placement. Three methods of validating keyboard input offer new programming possibilities. There are also five additional output options. You can easily format numeric data, position and duplicate text, and output hard copy of a screen on a serial printer.

Section III: More Programming Power

This is the stuff of programmers' dreams: sophisticated branching, condition testing and controlling loops, reusing variables within a program, structured programming to replace GOTO and GOSUB, and the ability to merge programs. At last we have the IF . . . THEN . . . ELSE statement, and its test conditions are even reusable by employing another Simons' BASIC statement—RCOMP ELSE. If you've been wondering about those "procedures" used in Pascal (and maybe listening to friends bragging about them), wonder no more. With Simons' BASIC you will code, call, and execute procedures, and even merge them with other programs!

Section IV: Low-Resolution Color and Graphics

Now you have an easy way to access border and screen colors without using pokes. Colors for backgrounds, characters filled with color, and flashing colors are all included. And your brilliant screen display can be saved and recalled. Each character in the Commodore character set can be easily redefined as your own special graphic character. These graphic characters can then be printed the same as standard characters.

Section V: High-Resolution Graphics

Another section for everyone! Here at last is easy graphics. Plot lines, rectangles, circles, arcs and angles on a high-resolution, 320 by 200 pixel, screen. Change the plotting colors and change them back again. Create your own shapes and fill them with color, change their background color, rotate, and flash them! You can also add a few characters or text, then output your screen display on a serial printer.

Section VI: Multicolor Graphics

Here you are introduced to the second graphic screen—the multicolor screen. The same plotting commands available in high resolution are also available in multicolor. However, multicolor allows three foreground colors instead of just one! There is, of course, a trade-off, and that is somewhat lower resolution. All differences between plotting multicolor graphics and high-reso-

lution graphics are covered. You will have a good idea of when to make the trade-off of color for the lower resolution.

Section VII: Sprite Graphics

Here are those creative, magical movable objects, dear to the hearts of all programmers. These are the critters that give programs that extra sophistication and excitement, whether it's an educational or game program. This section shows how high-resolution and multicolor sprites are designed, colored, displayed, and moved. It also shows how they can be controlled, made to collide, and removed! Programming details are covered for using sprites on all three Simons' BASIC screen types—low resolution, high resolution, and multicolor.

Section VIII: Simons' Sound

Here again is the same ease of use for the Commodore 64's music synthesizing capabilities as for the creation of sprites. Except when using the pulse waveform, rather than having to POKE registers, Simons' BASIC will do it for you! And even when using the pulse waveform, the Simons' BASIC function, SOUND, makes this very easy. All of this is accomplished with just six new statements. You have the use of your computer's three voices, four waveforms, and numerous sound envelopes. If you don't choose to make music, you can create sound effects for your game programs.

Section IX: Leaving the Keyboard

Here are four functions to assist you in communicating with your computer without using the keyboard. Read functions are provided for joystick, paddles, and the light pen.

As you use these commands in the programs throughout this book, the commands come to life and will become a part of your BASIC vocabulary. Each and every Simons' BASIC command is covered in this handbook.

After completing Section 1, Chapter 1, feel free to first browse through the book. You may wish to begin with the topic that most interests you. Programming aids covered in Chapter 1 are used throughout this book because they eliminate or reduce programming "dog work." If you choose to skip around in the book, be sure to first read that section's Introduction. Each introduction gives an overview of the commands covered in each of the section's chapters and notes if they are dependent upon one another. Since some statements build on knowledge of statements covered in a previous chapter, some chapters must be read in chapter order.

This book assumes a moderate knowledge of Commodore BASIC, upon which the Simons' BASIC language is built. You do not, however, need to be an expert programmer. A little programming experience is sufficient.

In browsing through this book you will notice several helpful aides in learning Simons' BASIC. The first, of course, is the organization of the book, with each section covering a specific type of programming. In addition, I have included an "illustrated commands" column with all programs demonstrating

more than one command. This enables you to flip through the book and easily see which programs use the command you are currently interested in. There is also an extensive Index, as well as the Simons' BASIC Command Reference Listing—Appendix A—for quick reference of each command, statement, and function. Appendix B lists all of the tables and grids used throughout the book, which you may wish to reference.

This book will provide many program ideas, as well as all of the information you need to use Simons' BASIC well. So, sit back, relax, and enjoy a new programming experience!

Limits of Liability and Disclaimer of Warranty

The author(s) and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author(s) and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author(s) and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Registered Trademarks

Commodore 64, Commodore 64 User's Guide, Simons' BASIC cartridge and Simons' BASIC User Guide are trademarks of Commodore Business Machines, Inc.

1 Programming Aids We All Need

Introduction

The following is a brief overview, by chapter, of the 24 Simons' BASIC programming aid commands. All of the command words are reserved words, and cannot be used in variable names.

Chapter 1

AUTO provides computer-generated line numbers, using a user-specified starting line number and line number increment.

RENUMBER will resequence the line numbers of the program in memory, using a user-specified starting line number and line number increment.

PAGE enables the user to control the number of program screen lines displayed by the LIST command. Rather than listing an entire program, or listing specific statement lines, the program can be listed by "pages", each containing a user-specified number of lines.

DELAY and the SHIFT key are used to control the speed at which a listing will be printed on the screen.

OLD will retrieve a program cleared from memory by the NEW command.

COLD clears memory and returns the system to the Simons' BASIC title screen.

Chapter 2

TRACE displays each line number of a program as each statement line is executed, allowing you to follow the sequence of operation of the program.

RETRACE causes tracing to resume after stopping and, if necessary, editing a program. The last set of line numbers executed before the program was stopped is again displayed, and tracing is resumed when the program is rerun.

DUMP displays the value of all non-array variables. This command may be used when a program is stopped either by using the RUN/STOP key or by a program terminator. This command may also be used as a program statement.

FIND searches a program for a user-specified code or character string, and displays the line numbers of each occurrence.

PAUSE halts program execution for a user-specified number of seconds.

ON ERROR is used to trap any one of 19 specific programming errors. The error number and the line number in which the error occurred are stored in the variables ERRN and ERRLN.

NO ERROR permits the display of the error number held in ERRN and the line number, held in ERRLN, in which it occurred.

Chapter 3

DISAPA: is used to specify that the code in a statement line is to be hidden.

SECURE hides all program lines beginning with the DISAPA: command. These lines then cannot be listed, providing program security of private or confidential code.

Chapter 4

The KEY command enables you to assign your own BASIC or Simons' BASIC commands to the Commodore 64's function keys. Using the Commodore logo key and CMDR with SHIFT, gives you eight more function keys, for a total of 16 programmable keys.

DISPLAY enables you to list the commands currently assigned to the function keys.

Chapter 5

DIV returns the quotient resulting from a division of two positive, integer numbers.

MOD returns the remainder of a division of two positive, interger numbers.

FRAC returns the fractional part of a number resulting from a numeric expression.

% converts a number from binary to decimal form.

$ converts a number from hexadecimal to decimal form.

EXOR performs an exclusive OR (XOR) between two numbers.

Chapter 6

DISK provides simplified disk entry, eliminating the need to specify a logical file number, device number, and secondary address.

DIR provides a user-specified partial listing, or a complete listing, of a disk directory.

Key to Entering Program Listings

Keying conventions used in this section and throughout the book are:

"[CLEAR]" designates SHIFT plus CLR/HOME
CMDR designates the Commodore logo key
"[DOWN]" designates the CRSR down key

If a symbol is to be repeated, the number of repetitions will be shown following the key, for instance, "[DOWN 2]".

Additional key conventions needed in other sections will be listed in that section's Introduction.

1 Statement Numbers and Program Listings

The program that will be used throughout this chapter, "Numbering and Listing," will demonstrate the AUTO, RENUMBER, and PAGE commands. If during this portion of the chapter you want to experiment with programs of your own, SAVE the Numbering and Listing program first, and LOAD it when you continue on in the chapter. The DELAY, OLD, and COLD command demonstrations may use either the chapter program or a program of your own.

AUTO

Simons' BASIC still uses line numbers; however, now they can be automatically generated by the AUTO command. You choose both the starting line number and the line number increment. The starting line number specified will be the first line number generated, and each succeeding line number will be increased by the increment specified. After the computer prints each line number, you enter only the statement(s) for that line.

Command format:
AUTO starting line number, increment

The upper limit for increment is 255. Using an increment greater than 255 will give the error message ?ILLEGAL QUANTITY ERROR.

There is *no default* for either starting line number of increment. *Both parameters must be used with the AUTO command.* If they are not used, the error message ?SYNTAX ERROR will be given. This may seem to be a limitation. However, the KEY command can be used to store the AUTO command with *your* preferred parameters, enabling you to use this preset AUTO command by simply pressing a function key. The KEY command is explained and demonstrated in Chapter 4.

To demonstrate the AUTO command,

ENTER: `AUTO 100,10`

The computer printed READY, then the line number 100 followed by the cursor. The computer is waiting for a statement line to be entered.

Now you will begin entering the demonstration program shown below. For guidance in entering the program, the line numbers are shown in parentheses to the left of the program listing. Remember—these line numbers are generated by the computer, and not typed by the user.

After entering the first statement line, the second line number, 110, will be printed. Continue to enter each statement line, in program order, next to the computer-generated line number.

Numbering and Listing

```
(100) PRINT "[CLEAR]":PRINT "TESTING AUTO":PRINT
(110) PRINT "LINE NUMBERS GENERATED BY THE":PRINT
      "AUTO COMMAND ARE:"
(120) FOR I=100 TO 230 STEP 10
(130) PRINT I;
(140) NEXT I
(150) PRINT "[DOWN 2]"
(160) PRINT "LINE NUMBERS GENERATED BY THE":PRINT
      "FIRST RENUMBER COMMAND ARE:"
(170) FOR I=0 TO 360 STEP 20
(180) PRINT I;
(190) NEXT I
(200) PRINT "[DOWN 2]"
(210) PRINT "PAGE 5 WILL BREAK THE PROGRAM LISTING"
(220) PRINT "AFTER LINE NUMBERS: 100, 190, 280, 400,
      490 AND 520."
(230) END
```

After you have entered the last program line, simply press RETURN

to cancel the AUTOmatic line numbering. Line number 24Ø cancels itself when entered.

RUN and LIST the program.

The output from the first FOR NEXT loop shows the statement numbers generated by the AUTO command. Each line number was increased by 1Ø as specified in the second parameter, increment.

Should you want to change the automatic line numbering while entering a program, do the following:

1. Press RETURN to cancel AUTOmatic line numbering, the same as you would at the end of your program.
2. Use the AUTO command again to specify the next statement number and the increment desired, and continue entering your program.

As you know, the maximum line number that can be used on the Commodore 64 is 63999. This limit is the same when using Simons' BASIC. If this line number is exceeded while using the automatic line numbering, the error message ?SYNTAX ERROR will be displayed.

Now you will seldom have to enter line numbers again!

RENUMBER

We often expand a program shortly after it's "finished" because we have thought of something that our program really can't live without. Unfortunately, this sometimes means that we don't have enough free line numbers where they're needed. The next time this happens to you, RENUMBER *may* be able to come to the rescue. However, always consider the following before using this option.

> **Use RENUMBER With Caution**
> RENUMBER only renumbers program lines. It *does not renumber line references used in program statements.* Using RENUMBER will leave all of your GOTO, ON GOTO, GOSUB, ON GOSUB, CGOTO, ON KEY, RETURN, RESTORE, and line referenced IF THEN and IF THEN ELSE statements incorrectly referenced in your program.

If you have no line number references in your program statements, or have only a few and it's worth changing them after renumbering, then RENUMBER will be useful.

Command format:
RENUMBER starting line number, increment

As with AUTO, there is *no default* for either starting line number or increment. *Both paraments must be used with RENUMBER.* Not using them will cause the error message ?SYNTAX ERROR to be displayed.

Use the Numbering and Listing program to test RENUMBER. LIST the program, then

ENTER: `RENUMBER 100,20`

LIST the program again.

Because the same starting line number was used, the first statement line remains 100. However, since a different increment was used, line 110 is now line 120, line 120 is now line 140, and so on, with each line number incremented by 20 rather than 10.

The renumbered program, shown below, will run exactly the same after being renumbered.

Numbering and Listing

```
(100) PRINT "[CLEAR]":PRINT "TESTING AUTO":PRINT
(120) PRINT "LINE NUMBERS GENERATED BY THE":PRINT
      "AUTO COMMAND ARE:"
(140) FOR I=100 TO 230 STEP 10
(160) PRINT I;
(180) NEXT I
(200) PRINT "[DOWN 2]"
(220) PRINT "LINE NUMBERS GENERATED BY THE":PRINT
      "FIRST RENUMBER COMMAND ARE:"
(240) FOR I=100 TO 360 STEP 20
(260) PRINT I;
(280) NEXT I
(300) PRINT "[DOWN 2]"
(320) PRINT "PAGE 5 WILL BREAK THE PROGRAM LISTING"
(340) PRINT "AFTER LINE NUMBERS: 100, 190, 280, 400,
      490 AND 520."
(360) END
```

RUN the program. The output from the second FOR NEXT loop shows the line numbers generated by the RENUMBER command.

When using RENUMBER, be careful to **avoid exceeding the maximum line number of 63999.**

Note: The RENUMBER command does not check for the line number limit of 63999 and will attempt to RENUMBER beyond this limit. For example, when testing this command with a program containing six statement lines and using RENUMBER 63000,1000, the line numbers generated were: 63000, 64000, 65000, 464, 1464, and 2464. The program did run correctly and could be saved, BUT the program **could not be reloaded!** The largest line number that BASIC can use is 65535, but the largest that can be altered from BASIC is 63999. The number 464 was generated by subtracting the total bytes available on the Commodore 64—65536 (64K)—from the next line number to be generated—66000—i.e., 66000-65536=464.

Now let's look at what would happen if this program contained a statement with a line reference. Add the following statement line to the program in memory. LIST the program and

ENTER:
```
290 IF I=380 GOTO 360
```

Since the value of I will be 380, line 290 will cause a branch to the END statement.

RUN the program.

Now RENUMBER and LIST the program again.

ENTER:
```
RENUMBER 100,30
LIST
```

Read the screen listing to "see" the effect of RENUMBER! The listing is also shown below.

Numbering and Listing

```
(100) PRINT "[CLEAR]":PRINT "TESTING AUTO":PRINT
(130) PRINT "LINE NUMBERS GENERATED BY THE":PRINT
      "AUTO COMMAND ARE:"
(160) FOR I=100 TO 230 STEP 10
(190) PRINT I;
(220) NEXT I
(250) PRINT "[DOWN 2]"
(280) PRINT "LINE NUMBERS GENERATED BY THE":PRINT
      "FIRST RENUMBER COMMAND ARE:"
(310) FOR I=100 TO 360 STEP 20
(340) PRINT I;
(370) NEXT I
```

```
(400)  IF I=380 GOTO 360
(430)  PRINT "[DOWN 2]"
(460)  PRINT "PAGE 5 WILL BREAK THE PROGRAM LISTING"
(490)  PRINT "AFTER LINE NUMBERS: 100, 190, 280, 400,
       490, AND 520."
(520)  END
```

RUN the program.

The result is: ?UNDEF'D STATEMENT ERROR IN 400. The statement line number referenced by line 400 is line number 360, the IF GOTO statement. This line number no longer exists due to renumbering. To correct this code, list the program and edit line 400 as shown below.

```
400 IF I=380 GOTO 520
```

The program will now execute correctly, branching from line 400 to the END statement.

Remember to be careful when renumbering!

PAGE

This command enables you to control the number of program screen lines (*not the number of statement lines*) displayed, breaking your program listing into "pages" of output. You must specify the number of screen lines you wish shown at one time. Each new "page" replaces the previous "page" shown on the screen. PAGE will output the number of screen lines specified plus one, as discussed below.

Command format:
PAGE number of lines

The *parameter*, number of lines, *must be used*, or the error message ?SYNTAX ERROR will be given. For the PAGE command to work properly, the maximum number that can be specified for the number of lines parameter is 22.

The following will cause a page length of 6 screen lines to be printed each time the RETURN key is pressed. Each "page" is discussed below.

ENTER:
```
PAGE 5
LIST
```

The first line number in your program is displayed: number 1ØØ. Press RETURN.

A new "page" appears. This listing shows the statement for the previously listed line number (1ØØ), the next five screen lines of the program, and the number of the first statement that will be shown on the next "page".

Continue to press RETURN to view the next "page" of the program. Each "page" will include: the statement(s) for the line number shown on the previous "page", the next 5 screen lines of your program, and the first line number of the next "page". Of course, the last "page" will contain the 1 to 6 screen lines remaining in the program.

Paging is turned off by using the PAGE Ø command.

```
ENTER:  PAGE Ø
        LIST
```

Now the program listed in the normal manner.

It is important to remember that screen lines—NOT statement lines—are the criteria for paging. For the program in memory, the command PAGE 3 will demonstrate this when displaying the fifth and sixth pages.

```
ENTER:  PAGE 3
        LIST
```

The first page will show line number 1ØØ. Press RETURN.

The second page will show statement lines: 1ØØ, 13Ø (two screen lines), and 16Ø. The line number 19Ø will also be printed. Press RETURN.

The third page will show statement lines: 19Ø, 22Ø, 25Ø, and the first screen line of 28Ø. No line number will be printed for the first statement line of the next "page", because this page ends with an incomplete statement line. Press RETURN.

The fourth page will show the second screen line of statement line 28Ø, and statement lines 31Ø, 34Ø, and 37Ø. The line number 4ØØ will also be printed. Press RETURN.

The fifth page will show statement lines: 4ØØ, 43Ø, and 46Ø (two screen lines). Statement number 49Ø will also be displayed. Press RETURN.

The sixth and last page will show statement lines 49Ø (two screen lines) and 52Ø, completing the program listing.

Of course you will normally be using paging with long programs, holding a full twenty lines on the screen at a time for program review

and/or debugging. For me, this is what makes the PAGE command very useful. However, you cannot edit program lines printed on the screen with the PAGE command. PAGE is for display only. To edit program lines you must end the PAGE listing, using the RUN STOP and RESTORE keys, in order to return the cursor. Then turn off paging with PAGE Ø and LIST—the old-fashioned way—those program lines that need editing. (Note: Paging need not be turned off to edit statements. However, if paging is not turned off, you can only list and edit one page of program screen lines, the number of which was specified in the last PAGE command. If there are statements to be edited in several different parts of the program, this is inconvenient and PAGE Ø should be used first.)

DELAY and the SHIFT Key

This is yet another way to control the printing of a program listing. This command is entered prior to the LIST command. The SHIFT key and the parameters of the DELAY command are used to control listing speed. When the SHIFT key is held down during program listing, the printing speed is controlled by the DELAY speed specified. The range of delay print speed is 1 to 255, with 1 providing the least delay and 255 providing the maximum delay speed.

Command format:
DELAY print speed number

The parameter, print speed number, must be used. There is no default. If a print speed number is not used, the error message ?SYNTAX ERROR will be given. If a print speed greater than 255 is specified, the error message ?ILLEGAL QUANTITY ERROR will be given. To test this command at the slowest speed on the program in memory,

ENTER:
```
DELAY 255
LIST
```

and immediately hold down the SHIFT key.

Each time you release the SHIFT key, your program will resume listing in the usual manner. In this way you can control the listing speed of portions of the program, or all of the program. This is similar to using the CTRL key to slow listing speed with Commodore BASIC. However,

DELAY 255 slows listing speed to a much greater extent. (Note: The parameter 0 slows listing speed to the same extent as 255.)

If DELAY is a useful command for you, test various numbers to find the print speed that you find most useful and make a note of it on the Simons' BASIC Command Reference Listing (Appendix A). After learning the use of the KEY command, you may want to assign this DELAY speed to a function key.

OLD

This is a super fail-safe command for all of us! I doubt that there is a programmer who has never once entered the NEW command and not instantly regretted it. The OLD command can save us from this depressing error by returning the program cleared by NEW. However, you have to use the OLD command **BEFORE ENTERING EVEN ONE NEW PROGRAM STATEMENT LINE**. Once a new program line is entered, the previous program can no longer be retrieved.

Command format:
OLD

Try out this life-saver command now.

```
ENTER:    LIST
          NEW
          LIST
          OLD
          LIST
```

The program was cleared from memory by NEW, and recalled by the OLD command.

COLD

The COLD command clears memory and returns the system to the Simons' BASIC title screen. Be patient after entering this command, because it takes several seconds to reset the computer to the start of Simons' BASIC. Those of you who purchased Simons' BASIC on tape or disk will love this software reset, preserving Simons' BASIC so that you don't have to reload it.

Command format:
COLD

Again, you can use the OLD command to restore your previous program IF NO NEW PROGRAM STATEMENT LINE HAS BEEN ENTERED. To try this command:

```
ENTER:  COLD
        LIST
        OLD
        LIST
```

Now enter either NEW or COLD and you'll be ready for Chapter 2.

2 Debugging Aids

This chapter will use several demonstration programs. The first program will be used to demonstrate five individual commands: TRACE, RETRACE, DUMP, FIND and PAUSE in the immediate mode. A second program will demonstrate the combined use of TRACE, PAUSE, and DUMP as program statements. Used as statements, their effectiveness and versatility as debugging tools are greatly increased. A third program will be devoted to the error trapping and display commands ON ERROR and NO ERROR.

TRACE

This command provides a computer-generated program walk-through, displaying each program line number in a screen window as the line is executed. This can be useful for studying program flow to check for possible branching or logic errors. But I must warn you that you have to be a very quick reader to follow this line number listing.

TRACE produces a window in the upper right-hand corner of your screen, showing a scrolling listing of the line numbers executed as the program runs. This listing shows a maximun of six line numbers at any one time. Scrolling and output can be slowed down somewhat by holding down the Commodore logo key.

Command format for turning on TRACE:
TRACE 1Ø

The following program will be used to demonstrate how TRACE performs. Line numbers are shown with the program either for user entry or, if using automatic line numbering, for reference. To use automatic

line numbering, first enter AUTO 10,10 then the program lines as demonstrated earlier in this chapter.

ENTER:
```
10 PRINT "[CLEAR]"
20 PRINT "PAYROLL FOR PART-TIME
EMPLOYEES"
30 PRINT "DEPT. #22, W/E 5/22/88"
40 PRINT:PRINT "EMP #","HOURS","PAY":PRINT
50 FOR I=1 TO 6
60 READ EN,H
70 PRINT EN,H,"$"H*4
80 TH=TH+H
90 NEXT I
100 PRINT:PRINT "TOTAL",TH,"$"TH*4
110 DATA 11,23,12,28,13,30,14,25,15,18,
16,17
120 END
```

This program will read the data employee number (EN) and hours worked (H) for six part-time employees paid at the rate of $4.00/hour. The output will be a very modest payroll listing for this department, along with total department hours and total pay. To turn on TRACE, RUN the program and slow the trace listing:

ENTER:
```
TRACE 10
RUN
```

and **immediately** hold down the Commodore logo key.

Each program line number appeared in the TRACE window as it was executed. Figure 2.1 (page 16) shows your screen output after program execution, with the TRACE window showing the last six statement lines executed.

Note: The format of the output from this program will be greatly improved in Section 2, Chapter 6, by using the Simons' BASIC command, USE, that allows formatting of lists of numbers.

Now we'll edit this program to incorrectly put the hours accumulator (TH) outside the READ loop, and RUN the program with TRACE again.

ENTER:
```
80
95 TH=TH+H
RUN
```

and **immediately** hold down the Commodore logo key to slow the TRACE output.

As you can see, statement line 95 was read only once. Therefore, the value of TH is the last value of the variable H.

TRACE has served its purpose. We will leave this incorrect code in the program to use with the next Simon's BASIC command, RETRACE.

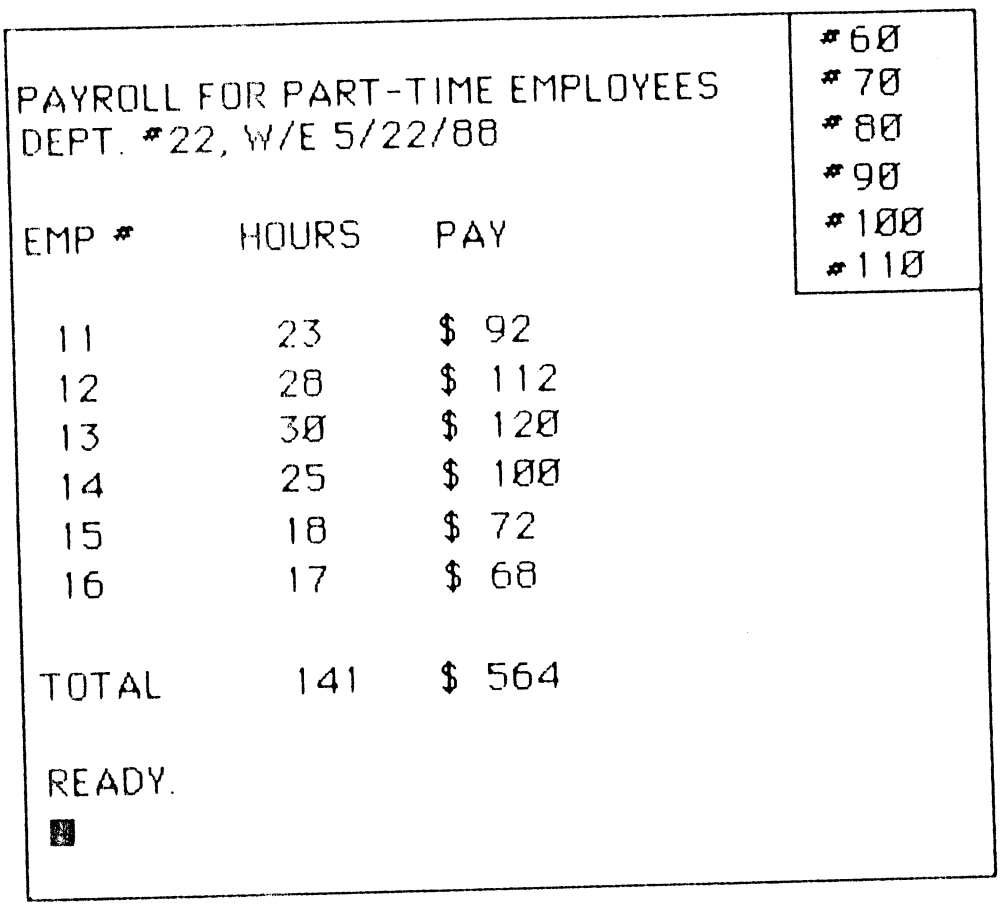


Figure 2.1: Use of TRACE Window.

Until TRACE is turned off, the TRACE window will be displayed each time the program is run. Turning TRACE off is a simple matter, as shown below.

Command format to turn off TRACE:
TRACE 0

When TRACE is turned off, the TRACE window disappears and the program executes normally.

```
ENTER:    TRACE Ø
          RUN
```

TRACE has one "bug" that I have found, causing it to interfere with program operation under certain conditions. TRACE will not perform correctly in a program containing the three BASIC statements "[CLEAR]", INPUT, and IF THEN. Such a program will only run correctly if the "[CLEAR]" statement is removed, and the screen is not cleared prior to running the program. This "bug" will be demonstrated in Chapter 3, page 30, using a program with this particular set of conditions.

RETRACE

If TRACE is on, the RETRACE command will generate the last TRACE window displayed for the previous execution of the program. The program could have ended either by using the RUN/STOP key or a program terminator (STOP or END). Before RETRACE is entered, the program may be edited and/or listed, and the screen cleared.

Command format:
RETRACE

When the following commands are entered, TRACE will be turned on and the program will be RUN. Make a note of the final six line numbers that appear in the TRACE window. These are the line numbers that will be shown when RETRACE is executed.

```
ENTER:    TRACE 1Ø
          RUN or RUN and press RUN/STOP
```

Now LIST and edit the program and execute RETRACE, as shown below. The edited statements will change the program back to correctly accumulate total hours in variable TH by moving it back inside the I loop.

```
ENTER:    LIST
          8Ø TH=TH+H
          95
          RETRACE
          RUN
```

Hold down the CMDR key to slow printing of the output.

The TRACE window showed the last six statements executed during the previous program run, before tracing for this run began.

Clear the screen. To execute the program again with TRACE on after executing RETRACE and clearing the screen, you must **move the cursor under the TRACE window, then enter RUN.**

If RUN is entered in the normal fashion, you will receive the error message ?UNDEF'D STATEMENT ERROR. (However, if after receiving this error message you enter RUN again, both your program and TRACE will run correctly—without another error message.)

If RETRACE is used after turning TRACE off (TRACE Ø) you will receive the error message ?BAD MODE. Remember, RETRACE can only be used when TRACE is on.

Move the cursor under the TRACE window, and

ENTER: RUN

The program ran correctly, again "tracing" the program statements in the TRACE window.

Now turn TRACE off.

ENTER: TRACE Ø

DUMP

This command lets you print the values of all non-array variables used in a program. The values shown will be those contained in each variable when the DUMP command is executed. DUMP can be executed following a program break by using the RUN/STOP key, by a programmed STOP, or after the program is run.

Command format:
DUMP

Adding a character string variable to the program in memory will provide a good assortment of variables to demonstrate DUMP.

ENTER:
```
15 C$="SPEEDY DATA CO."
95 PRINT "OUTPUT BY: ";C$
```

RUN the program now in memory, then

ENTER: DUMP

The screen display shows:

```
C$="SPEEDY DATA CO."
I=7
EN=16
H=17
TH=141
```

As you will see later in this chapter, this command is also very useful when used as a program statement.

FIND

This command is used to search for specific code or character string occurrences in a program. All program line numbers in which the FIND code or character string occurs are displayed.

Command format:
FINDcode
or
FIND"character string"

There cannot be any space between the command FIND and the code or character string given. A space will not cause an error message, but FIND will not execute.

LIST the program in memory. FIND will be used to locate and display the line numbers for each occurrence of: (1) the numeric variable TH, (2) the BASIC word DATA, (3) the character string "HOURS", and (4) the character string variable C$.

```
ENTER:   FINDTH
         FINDDATA
         FIND"HOURS"
         FINDC$
```

You see that TH occurs in statement lines 8Ø and 1ØØ, the DATA statement is in line 11Ø, "HOURS" is in line 4Ø, and C$ occurs in statement lines 15 and 95.

In a long program, FIND can be a super time-saver. If you wish to change a variable name (perhaps a reserved word crept in), add data values, change a character string, and so forth, FIND will conduct a free search for those statement lines.

PAUSE

This command allows you to halt program execution for a specified number of seconds. PAUSE has many applications. We will look at several in this chapter, as well as in Section VI, Simons' Graphics.

Command format:
PAUSE number of seconds
or
PAUSE "message",number of seconds

When using the message option, the message must be enclosed in quotation marks and separated from the parameter, number of seconds, by a comma.

For a quick demonstration of PAUSE, add the following statement lines to the program in memory.

ENTER:
```
17 PAUSE "5 SECOND PAUSE",5
93 PAUSE 10
```

The second pause of 10 seconds will seem quite long! LIST and RUN the program.

Combining TRACE, DUMP, and PAUSE

The following program demonstrates the use of these three commands as program statements with PAUSE allowing DUMP and TRACE to be used more effectively. By using PAUSE with DUMP we are able to read the value of program variables during program execution. PAUSE will also halt TRACE for a longer look at the sequence of lines executed.

To save keying time, you may enter the following program using automatic line numbering by first entering AUTO 10,10. If you use AUTO, do not copy the line numbers in the program listing. The three commands illustrated by this program are shown to the left of the program listing.

Commands Illustrated	Program Listing
TRACE	10 TRACE 10
	20 A=6
	30 B=7
	40 FOR I=1 TO 9 STEP 3
DUMP	50 DUMP
	60 PRINT A+B+I
PAUSE	70 PAUSE 5
	80 NEXT I
	90 END

RUN the program.

Dumping variable values within the I loop while tracing program execution and pausing the program to read the results of both DUMP and TRACE show exactly what is happening in the program. Not only is this an excellent debugging aid, but it's an excellent tool for learning BASIC programming. Often "seeing is believing" is very helpful to a learner.

ON ERROR and NO ERROR

Error trapping is provided by the ON ERROR command. This command allows you to avoid a "program crash" by branching to another part of the program when an error is detected. After disabling error trapping, using NO ERROR displays both the error number (ERRN) and the number of the statement line in which the error occurred (ERRLN). Figure 2.2 lists the errors, by error number, that can be trapped by Simons' BASIC.

ERROR NUMBER	ERROR DESCRIPTION
1	Too many files
2	File open
3	File not open
4	File not found
5	Device not present
10	NEXT without FOR
11	Syntax
12	RETURN without GOSUB
13	Out of Data

```
14          Illegal quantity
15          Overflow
16          Out of memory
17          Undefined statement
18          Bad subscript
19          Re-dimensioned array
20          Division by zero
21          Illegal direct
22          Type mismatch
23          String too long
```

Figure 2.2

Statement format:
ON ERROR:GOTO line number

This command is used either at the beginning of a program, or before that portion of a program where error trapping is required. The GOTO branches to the statement line containing the NO ERROR command and/or the PRINT statement for ERRN (error number) and ERRLN (line number in which the error occurred), as shown below.

To turn off error trapping and print the error and line number found, use:

Command format:
NO ERROR:PRINT ERRN;ERRLN

The PRINT statement following the NO ERROR command can, of course, include text and use a different separator. ERRN will contain the error number, and ERRLN will contain the line number in which the error occurred.

The following program will trap five errors, one at a time, as they occur in the program. The errors are: illegal quantity, syntax, out of data, NEXT without FOR, and RETURN without GOSUB. After each error is trapped and the error and line number printed, the incorrect statement will be edited, and the program rerun to trap the next error. This process will be repeated until all five errors have been found and corrected.

To use automatic line numbering with the following program, enter AUTO 10,10 before entering the program. Line numbers are shown for program clarity and ease of program entry, and for anyone who chooses not to use automatic line numbering. The command illustrated is shown to the left of the program listing, and the error numbers are shown in parentheses to the right of the statements containing the error. Note that statement number 10 will cause the program to branch to statement line 150 when an error is encountered. Statement line 150 will turn off error trapping (NO ERROR) and print the error number (ERRN) and line number in which the error occurred (ERRLN).

Command Illustrated	Program Listing	Error #
ON ERROR	10 ON ERROR:GOTO 150	
	20 PRINT "[CLEAR]" TAB(520) "TITLE IS ERROR"	(14)
	30 GOSUB 120	
	40 FOR I=1 TO 5	
	50 READ D	(13)
	60 FOR J=1 TO 3	
	70 PRIT I,J,D	(11)
	80 NEXT I	
	90 GOSUB 120	
	100 NEXT J	(10)
	110 DATA 15,25,35,45	
	120 REM SPACING ROUTINE	
	130 PRINT:PRINT	
	140 RETURN	(12)
NO ERROR	150 NO ERROR:PRINT "ERROR #";ERRN;	
ERRN	"IN LINE"; ERRLN	
ERRLN		

RUN the program. On encountering an error, the GOTO statement in line 10 sent the program to line 150 to turn off error trapping and print the error number and the line number in which it occurred.

The error message reads: ERROR #14 IN LINE 20. LIST line 20. The quantity given for TAB is an illegal quantity, since it exceeds 255. Edit line 20 to read:

```
20 PRINT TAB(52) "TITLE IS ERROR"
```

This statement line will cause the title to be printed in the center of the second screen line.

RUN the program again to trap the next error.

The error message reads: *ERROR #11 IN LINE 7Ø*. LIST line 7Ø. The syntax error is obvious; PRINT is misspelled.

Edit line 7Ø to correct the misspelling:

```
7Ø PRINT I,J,D
```

To locate the next error, enter RUN.

This error message reads: *ERROR #13 IN LINE 5Ø*. LIST line 5Ø. There is either a missing DATA statement or insufficient data values in the DATA statement. To find out which is the case:

```
ENTER:    FINDDATA
```

The response is 11Ø. LIST line 11Ø and add the value 55 to the end of the statement line. This statement should now read:

```
11Ø DATA 15,25,35,45,55
```

On to the fourth error! Enter RUN.

The error message reads: *ERROR #1Ø IN LINE 1ØØ*. LIST the program. The problem is crossed loops. Lines 8Ø and 1ØØ must be edited as follows:

```
8Ø NEXT J
1ØØ NEXT I
```

Now for the last error in this program. RUN the program.

The error message reads: *ERROR #12 IN LINE 14Ø*. There is no division between the main program and the subroutine. To correct this, enter the following statement line.

```
115 END
```

When the program is error free, ERRN will contain the number 128—meaning *No Error*, and ERRLN will contain the line number of the last statement line read.

RUN the program.

The message displayed is: *ERROR #128 IN LINE 115*, telling you that there are no further errors that Simons' error trapping can detect in the program.

Since the program is now error free, remove the error trapping and display statements.

ENTER:
```
10
150
```

RUN the error-free program.

Of course, each of your programs will have a different error or errors needing trapping. Fortunately, Simons' BASIC will be a very big help!

This ends our tour of Simons' debugging aids. Clear computer memory for Chapter 3.

NOTE: The OUT command is supposed to disable the last ON ERROR command. However, this command does not always function properly and in certain circumstances causes a system crash, losing the program in memory. Therefore, the OUT command is not covered in this book. It should be noted, however, that the powerful ON ERROR and NO ERROR commands perform error trapping excellently, without need for the OUT command.

3 Program Security Aids

This chapter will show you how to keep any portion of a Simons' BASIC program secret by hiding specific program code. The use of two Simons' command—DISAPA: and SECURE—will prevent the code from being listed.

Be sure to follow the caution of keeping an unsecured copy of each program that uses these security aids, because no one—*not even you*—will be able to read the code that has been SECUREd.

DISAPA:

This command specifies that all statements following DISAPA: on a program line are to be hidden. That line can then be secured, preventing the line from being listed as part of the program.

Statement format:
DISAPA: statements

The DISAPA: command automatically places four additional colons before a DISAPA: statement; making it necessary to allow space for them on the program line. Every statement line is, of course, limited to 80 characters including RETURN. For a DISAPA: statement line it's 75 characters, 80 characters less five characters—four characters for the Simons' inserted colons and one character for the RETURN key.

The following program will demonstrate the DISAPA: command. If you wish to use automatic line numbering enter AUTO 1Ø,1Ø before entering the program statements. Line numbers are shown in the pro-

gram listing for your reference or use if you choose not to use automatic line numbering. The command illustrated is shown to the left of the program listing.

Command Illustrated	Program Listing
	10 PRINT "[CLEAR]"
	20 A=A+1:IF A>3 GOTO 60
	30 INPUT "USER CODE";UC$
DISAPA	40 DISAPA:IF UC$="SIMONS" THEN PRINT "ENTRY GRANTED":GOTO 70
	50 PRINT "NO ACCESS":PRINT:GOTO 20
	60 PRINT "ENTRY DENIED"
	70 END

LIST the program.
Statement line 40 now reads:

```
40 DISAPA:::::IF UC$="SIMONS" THEN PRINT "ENTRY
GRANTED":GOTO 70
```

The four additional colons have been inserted.

Note: If a DISAPA statement line is edited on screen by using the editing keys, four additional colons will be inserted each time the statement line is screen edited. So be sure there will be enough space on the program line. If there's not enough space, retype the line.

SECURE

This command hides the code designated by the DISAPA: command, preventing the code from being listed.

Command format:
SECURE 0

Note: This command might be hard to remember because most Simons' BASIC commands using the parameter 0 turn off a command. You will just have to remember that SECURE is different.

As soon as this command is entered, you will no longer be able to list any statement line beginning with the DISAPA: command. Only the line number will show on the listing.

REMEMBER, when using these commands in your own programs, SAVE an unsecured back-up copy of the program.

To hide statement line 40 in the program in memory,

ENTER: `SECURE 0`

LIST the program.

The statements in line 40 will no longer list. The user code is now secured.

The program listing on your screen shows:

```
10 PRINT " ♥ "
20 A=A+1:IF A>3 GOTO 60
30 INPUT "USER CODE";UC$
40
50 PRINT "NO ACCESS":PRINT:GOTO 20
60 PRINT "ENTRY DENIED"
70 END
```

This demonstration program will give the user three opportunities to enter the correct user code. If the correct user code is not entered, access is denied. RUN the program. If you wish to test the program, first use your name for input, then use the user code: SIMONS.

RUN the program. The program runs normally.

These commands are invaluable for securing confidential information, such as employee salaries or student grades. Any number of statement lines may be secured by using DISAPA and SECURE. To demonstrate this, add the following program lines to the program in memory. To save keying in time, you may use AUTO 60,10 for the following statement lines.

```
60 PRINT "ENTRY DENIED":GOTO 120
70 PRINT:PRINT "SPEEDY DATA CO., DEPT. #22"
80 PRINT:PRINT "EMP #","SALARY":PRINT
90 DISAPA:READ EN,SAL
100 IF EN=-99 GOTO 120
110 DISAPA:PRINT:PRINT EN,SAL:GOTO 90
120 DISAPA:DATA 11,19500,12,14600,13,22000,14,31500,
15,18650,-99,-99
130 END
```

To SECURE lines 9Ø, 11Ø, and 12Ø,

ENTER: SECURE Ø

LIST and RUN the program.

The program runs correctly, and no unauthorized person can read the confidential salary information.

If you entered the user code SIMONS, your screen output is:

```
USER CODE? SIMONS
ENTRY GRANTED

SPEEDY DATA CO.,DEPT. #22

EMP #       SALARY

  11          19500

  12          14600

  13          22000

  14          31500

  15          18650

READY
```

Figure 3.1: Output of Security Program.

NOTE: When TRACE is used with a program containing a DISAPA: statement, TRACE will generate an additional line number, #14ØØ6, for each line containing a DISAPA: statement. This does not affect program operation or the operation of TRACE.

Before using TRACE with the program in memory, the "[CLEAR]" statement (1Ø) must be removed. As mentioned in Chapter 2, this program has the particular sequence of statements that will cause incorrect program operation when using TRACE. The sequence of statements is: "[CLEAR]", INPUT, and IF THEN. With TRACE on, the IF THEN statement (line 4Ø) will be ignored until after INPUT data is entered three times. The program will run correctly with TRACE on only if the clear/home statement is removed, and you **do not** clear the screen before running the program. For a demonstration of this "bug", turn on TRACE (ENTER: TRACE 1Ø) and RUN the program without removing statement line 1Ø. Then remove line 1Ø and RUN the program again. If you have a problem with the TRACE command and another program, try removing the clear/home statement if your program contains one.

This concludes our work with the Simons' BASIC program security aids, but not with this program. This program will be used to demonstrate the commands covered in the next chapter. If you are not going on to the next chapter immediately, SAVE the program now in memory for use with Chapter 4, Assigning Commands to the Function Keys.

4 Assigning Commands to the Function Keys

This chapter will use the security aids program from Chapter 3 to demonstrate the function key assignment command. As mentioned in Chapter 1, these commands eliminate the necessity for having a default option for Simons' BASIC commands. You can set your own preferred default options for commands, using the programmable function keys. Of course, you can also program these keys for other BASIC commands, such as LIST and RUN.

Simons' BASIC provides 16 function keys—8 more than Commodore BASIC—and all 16 keys are programmable. The method of accessing the 16 functions using the four Commodore function keys is shown in Figure 4.1. Note that the odd numbered functions, f1 through f7, only require the function key to be pressed, and the odd numbered functions f9 through f15 require pressing function keys f1 through f7 **and** the Commodore logo key. The even numbered keys are always accessed while pressing SHIFT; f2 through f8 uses the function key and SHIFT, while f10 through f16 uses the f2 through f8 function keys **and** the CMDR **and** SHIFT keys.

KEY

This is the command used to program a function key to issue a user-specified command. You choose the function key and the command that is assigned, as shown below.

Accessing the 16 Function Keys

Function Number	*f Key Only*	*f Key and SHIFT Key*	*f Key and CMDR Key*	*f Key and CMDR and SHIFT Keys*
1	f1			
2		f2		
3	f3			
4		f4		
5	f5			
6		f6		
7	f7			
8		f8		
9			f1	
1Ø				f2
11			f3	
12				f4
13			f5	
14				f6
15			f7	
16				f8

Figure 4.1

Command format:
KEY function key number,"command"

The function key number is, of course, any function key number , 1 through 16. The command can be any Simons' BASIC or Commodore BASIC command. The command **must** be enclosed in quotation marks. The length of the command is limited to 15 characters. Exceeding the 15-character limit will not cause an error message. Your command will simply be truncated to 15 characters! This is an easy mistake to catch, since you will be using DISPLAY to list your function key commands.

In case you cleared memory after the last chapter and didn't SAVE the Chapter 3 program, the program listing follows. I have not included the DISAPA: statements in this listing, since they do not affect demonstrating the KEY and DISPLAY commands. Secured statements will execute exactly the same. Those of you using the Chapter 3 program can check the code below to "read" the secured lines in your program.

```
10 PRINT "[CLEAR]"
20 A=A+1:if A>3 GOTO 60
30 INPUT "USER CODE";UC$
40 IF UC$="SIMONS" THEN PRINT "ENTRY GRANTED":GOTO 70
50 PRINT "NO ACCESS":PRINT:GOTO 20
60 PRINT "ENTRY DENIED":GOTO 120
70 PRINT:PRINT "SPEEDY DATA CO., DEPT. #22"
80 PRINT:PRINT "EMP #","SALARY":PRINT
90 READ EN,SAL
100 IF EN=-99 GOTO 120
110 PRINT:PRINT EN,SAL:GOTO 90
120 DATA 11,19500,12,14600,13,22000,14,31500,
15,18650,-99,-99
130 END
```

The following commands will program function keys f1, f2, f9, f10, f13, and f14.

```
ENTER:   KEY 1,"RUN"
         KEY 2,"PAGE 10"
         KEY 9,"LIST"
         KEY 10,"PAGE 0"
         KEY 13,"NEW"
         KEY 14,"OLD"
```

As soon as the function key is pressed, the command programmed for that key will appear on your screen. You need only press RETURN to enter the command.

Press f1 and RETURN to RUN the program.

Press SHIFT f2 and RETURN to enter the PAGE 10 command.

Now list the program by pressing CMDR f1 and RETURN. Continue pressing return until the entire listing prints.

Turn off the PAGE command by pressing SHIFT CMDR f2 and RETURN.

Again list the program by pressing CMDR f1 and RETURN.

Clear memory by pressing CMDR f5 and RETURN.

Return the program to memory by pressing SHIFT CMDR f6 and RETURN.

This certainly demonstrates the benefits of assigning commands to the function keys. However, they are even more useful when a carriage return is included as part of the function key command. This eliminates the need to press the RETURN key. To do this, add + **CHR$(13)** to the KEY command. (As you probably know, the function CHR$ takes the

PET ASCII number that is assigned and converts it to the character it represents. The PET ASCII number 13 is the RETURN key.)

Command format using the carriage return:
KEY function key number,"command" + CHR$(13)

Remember: The command assigned **must** be enclosed in quotation marks.

To change the f1 and f9 commands to include a carriage return,

```
ENTER:   KEY 1,"RUN" + CHR$(13)
         KEY 9,"LIST" + CHR$(13)
```

Press f1 to RUN the program, then CMDR f1 (for function 9) to LIST the program.

The function keys will remain programmed until the computer is turned off or the COLD command is entered. You can return a program to memory using OLD after the COLD command is entered, but you lose the function key commands. However, the functions assigned are not effected by clearing memory with NEW.

It's a good idea to write a short program to assign frequently used commands to the function keys and LOAD it whenever you use Simons' BASIC. Remember, once you're spoiled by using the function keys, it will be hard to go back to typing and entering commands!

DISPLAY

To display function key assignments, use

Command format:
DISPLAY

To review the function key assignments now in memory,

```
ENTER:   DISPLAY
```

The following function key assignments are shown on the screen:

```
DISPLAY
KEY1,"RUN"+CHR$(13)
```

```
KEY2,"PAGE 10"
KEY3,""
KEY4,""
KEY5,""
KEY6,""
KEY7,""
KEY8,""
KEY9,"LIST"+CHR$(13)
KEY10,"PAGE 0"
KEY11,""
KEY12,""
KEY13,"NEW"
KEY14,"OLD"
KEY15,""
KEY16,""
```

Before ending this chapter, add the following key command for automatic line numbering having a starting line number of 10 and incremented by 10.

```
ENTER:    KEY 5,"AUTO 10,10" + CHR$(13)
```

Most program listings throughout this book begin with statement line 10 and are incremented by 10 so that line numbers for these programs can be generated by the stroke of a key!

5 Added Math Functions

This chapter reviews the three new arithmetic functions available on Simons' BASIC—DIV, MOD, and FRAC, as well as the two numeric conversion functions—% and $, and EXOR—a function to provide the ability to perform an exclusive OR.

All of these functions may be used in both the immediate and program mode.

DIV

This function is used to return an integer **quotient** (result) of a division of two decimal numbers. These numbers are converted to floating point before the division is performed, and the remainder, when one exists, is ignored.

Function format:
DIV(x,y)

In the argument (x,y), x is the dividend and y is the divisor. Both x and y **must** be positive numbers.

Both of the following entries will produce the same result, since the remainder is ignored.

```
A=DIV(66,3):PRINT A
A=DIV(67,3):PRINT A
```

Each will return a value of 22 for A.

Remember, DIV can only use **positive numbers**. Using a negative

number for x or y, or both, will result in an ILLEGAL QUANTITY ERROR.

MOD

This function is used to return only the **remainder,** *not the quotient,* of a division of two positive whole numbers (integers).

Function format:
MOD(x,y)

In the argument (x,y), x is the dividend and y the divisor. The entry:

```
X=2173:Y=64:C1=MOD(X,Y):PRINT C1
```

would return the result 61 as the value of C1, the remainder of 2173/64. Taken one step further, IF C1>Ø then X is an odd number—often a necessary piece of knowledge.

Remember, MOD can only use **positive numbers**. Using a negative number for x or y, or both, will result in an ILLEGAL QUANTITY ERROR.

FRAC

This function returns the fractional part, up to a maximum of nine digits, of either a decimal number (internally converted to a floating point number) or a floating point number. An individual number or a numeric expression may be used.

Function format:
FRAC(numeric expression)

Both positive and negative numbers may be used in the numeric expression. For example:

`PRINT FRAC(-π)` returns the value -.141592653
`PRINT FRAC(-22/-6)` returns the value .666666667
`X=FRAC(2Ø8/3):PRINT X` returns the value .333333343 for X

%

This function converts a binary number into its decimal equivalent.

Function format:
%binary number

The following entry would return 147 as the value of A.

```
A=%10010011:PRINT A
```

If a non-binary number is used as the argument, the error message NOT BINARY CHARACTER will be given. For example, omitting one digit of a binary number—as in **1001001**—would result in this error message.

$

This operator converts a hexadecimal number into its decimal equivalent.

Function format:
$hexidecimal number

The following entry would return the decimal value of 49152.

```
PRINT $C000
```

If a non-hexadecimal number is used as the argument, the error message NOT HEX CHARACTER will be given. In the above example, if the letter O had been mistakenly used in place of the zeros—**$COOO**—the error message NOT HEX CHARACTER would be given.

EXOR

This operator performs an exclusive OR (XOR) between two numbers.

Function format:
EXOR(n,n)

The two numbers given in the EXOR argument are first converted into binary numbers. The two binary numbers are then compared bit by bit. If both bits are the same; that is, both are either Ø or 1, the corresponding bit is cleared (set to Ø). If the bits compared are different; that is, one is Ø and the other is 1, the corresponding bit is not cleared (set to 1). The value returned is the decimal equivalent of this bit by bit comparison.

All three entries shown below would return the decimal value 109.

```
X=%11111110:Y=%10010011:PRINT EXOR(X,Y)
PRINT EXOR(254,147)
PRINT EXOR(147,254)
```

Using the first entry shown above, the bit comparison is performed as shown below.

```
11111110
10010011
01101101  which equals decimal 109 (1+4+8+32+64)
```

All of these numeric aids, whether used singly or in combination, will be very useful in many programming situations.

6 Disk Handling

There are but two new disk commands available on Simons' BASIC, but both will certainly be welcomed by Commodore 64 programmmers. The new DISK command shortens disk entries by eliminating a logical file number, device number, and secondary address when opening a channel to a disk drive unit. It then closes the channel after the operation is executed. The new DIRectory command allows you to list specific portions or all of a disk directory. (**Note:** Some Simons' BASIC commands will not work correctly with DOS Wedge loaded, such as INSERT, INST, PLACE, and USE. Therefore, it would be best not to load Wedge when working with Simons' BASIC.)

DISK

This command opens a disk file, performs the operation specified, then closes the file.

Command format:
DISK "operation"

The parameter "operation" **must** appear within quotation marks.

This command replaces the following BASIC commands:

1. OPEN logical file number,device number,secondary address
2. PRINT #logical file number,
3. CLOSE logical file number

To format a new disk, with the name SIMONS, the entry would be:

```
DISK "NØ:SIMONS,Ø1"
```

Of course, the "operation" must still specify "N" for format new disk

followed by the disk drive number, separated by a colon from the disk name, and followed by the disk identification.

To erase—or "scratch"—a program named MAILER, the entry would be:

```
DISK "SØ:MAILER"
```

The "S", of course, specifies "scratch."

DIR

This command lists all, or specific parts, of the disk directory for the disk in device #8. The options for listing a portion of the disk directory are:

1. Files whose names begin with a particular character or characters.
2. Files where a specific character or characters are in a particular position within the file name.
3. A combination of the above two options.

Command formats:

DIR"$	to list the entire directory

This command replaces the BASIC commands LOAD "$",8:LIST.

DIR"$:character(s)*	option 1 above
DIR"$:?character(s)	option 2 above
DIR"$:?character(s)*	option 3 above

An example of option 1 would be to list all directory files beginning with the characters EMP. The DIR entry would be:

```
DIR"$:EMP*
```

A possible listing would include the files: EMPNO, EMPADD, EMPSAL, EMPOT, etc. In other words, all files beginning with the three characters EMP.

An example of option 2 would be a listing of all files ending with the characters DATA. You would, however, have to know how many letters precede the characters DATA, since **each** missing character must be represented by a question mark. If the files ending with DATA had four letters preceding DATA, the DIR entry would be:

```
DIR"$:????DATA
```

Your listing would include such files as: SALEDATA, ENGRDATA, ACCTDATA, and so forth.

An example of option 3 would be a listing of all files whose third and fourth letters are CO. Two question marks would be used to replace the first two letters, which could be any two letters. An asterisk following CO would indicate that more characters could follow. The DIR entry would be:

```
DIR"$:??CO*
```

All filenames whose third and fourth characters were CO would be listed.

II Added Input and Output Options

Introduction

Following is a brief overview, by chapter, of the 16 Simons' BASIC input and output commands. All of the command words are reserved words and cannot be used in variable names.

Chapter 7

CENTRE displays text centered in the middle of a screen line, without the user specifying the text length. This eliminates counting text or using LEN, and TAB or SPC.

INSERT enables the programmer to create an expanded character string, by *inserting* one string into another. This command saves programming time and tedious keyboarding by eliminating the need to type in new strings or struggle with the string extractors—LEFT$, RIGHT$, and MID$.

INST is used to overwrite a portion of a character string with another string, creating a new character string. This is another text time-saver for programmers.

PLACE searches for user-specified characters (a sub-string) within a specified character string (a main string), and, if found, returns the starting location of the sub-string. This command is useful both for searching strings for specific characters, and for returning their sub-string location. This sub-string location can then be used with both the INSERT and INST commands.

DUP provides an easy method of duplicating a character string a specified number of times, thereby creating a new, and larger, character string.

CSET allows you to switch between the two Commodore 64 character sets—the upper case/graphic character set and the upper/lower case character set.

INV causes character background to reverse; normal characters are displayed in reverse field and reverse field characters are displayed normally.

Chapter 8

INKEY is used to test for and store the value of a function key input from the keyboard. It is used as the numeric variable of an ON GOTO or ON GOSUB statement using the function key number input for branching. This is a valuable programming tool for menu-driven programs.

ON KEY is used for accepting a user-specified input character from the keyboard, and branching based upon the character input.

RESUME is a companion command for ON KEY, performing a branch back to the last ON KEY command performed.

FETCH allows the programmer to limit the program user's keyboard input by controlling the type (alphabetic or numeric) and number of characters accepted as input.

Chapter 9

AT is a PRINT function used to position text at a user-specified screen location using both row and column parameters. This eliminates the need to use the cursor control keys and the TAB or SPC functions to position text.

USE is the command that formats lists of numbers—aligns the decimal point. It's use is obvious for any type of financial reporting.

MOVE is used to duplicate a specified portion of the screen, and display the duplicate at a given screen location.

OPTION is the command used to print a program listing with all Simons' BASIC commands highlighted in reverse-field.

HRDCPY is the statement used to print hard copy of a low-resolution screen (the normal screen-type) on a serial bus printer.

Key to Entering Program Listings

Additional key conventions used in this section and throughout this book are:

"[HOME]" designates UNSHIFTED CLR/HOME

Graphic characters will be separated by a comma, and represented by either SHIFT or CMDR and a key.

If a character is to be repeated, the number of repetitions will be shown directly after the key.

7 Character String Input Options

This chapter covers the six Simons' BASIC methods used to position or alter character strings. When possible, these commands will first be demonstrated in the immediate mode. Then a program will be developed to accommodate each new command reviewed. Therefore, do not clear computer memory during this chapter without first saving the demonstration program. When you are ready to continue the chapter, LOAD the program back into memory.

CENTRE

Centering text on the screen is an easy task with this command. The only difficulty is remembering the English spelling of the command word. Remember, David Simons is a young Englishman!

Command format:
CENTRE"character string"
or
CENTRE string variable name

Statement format:
CENTRE"character string":PRINT
or
CENTRE string variable name:PRINT

For CENTRE to function correctly, the string cannot exceed 39 characters. If the string is longer than 39 characters, the string will be broken into two lines, with neither line centered. No error message will be given.

Because CENTRE does not force a carriage return, it must be followed by PRINT unless it is the last program statement containing a display command. If a carriage return is not used, the next character(s) displayed will begin at the end of the centered text. If a second CENTRE statement follows without using a carriage return, that character string will not be centered.

To display "SIMONS' BASIC" centered on a screen line, enter the following CENTRE command.

```
CENTRE "SIMONS' BASIC"
```

To demonstrate CENTRE statements, the following program will display a three-line centered address. If you wish to use automatic line numbering when entering this program, enter AUTO 10,10,—**or,** if your function keys are still programmed as they were in Chapter 4, use f5. Again, line numbers are shown for program clarity and ease of entry.

Program Listing

```
10 PRINT "[CLEAR]":PRINT
20 CENTRE "SIMONS' ARCADE":PRINT
30 CENTRE "1200 WILSON DRIVE":PRINT
40 CENTRE "WEST CHESTER, PA. 19380"
50 END
```

Run the program. Lines 20, 30, and 40 are centered on your screen. To demonstrate what happens when CENTRE is not followed by a carriage return, LIST the program, **remove the PRINT statement from line 20**, and add the following statement line.

```
ENTER:    45 PRINT "INVOICE NUMBER 1024"
```

RUN the program.

The CENTRE statement on line 30 was not performed properly because it was not preceded by a carriage return. The PRINT statement on line 45 begins at the end of the previous CENTRE character string, because it was not preceded by a carriage return.

As long as you remember to *include a carriage return*, either through PRINT alone or PRINT CHR$(13) followed by a semi-colon, CENTRE will work perfectly in your programs.

To demonstrate the use of CENTRE with a string variable, and add back the carriage returns, LIST the program and enter the following program changes.

```
20 A$="SIMONS' ARCADE":CENTRE A$:PRINT
43 PRINT:PRINT
```

RUN the program. The centered strings and the printed text are now displayed correctly.

Leave your computer on or save this program, since we will be adding to it throughout this chapter to demonstrate each chapter command.

INSERT

This statement allows you to expand a character string by *inserting* one string *into* another. It does not insert characters at the beginning or end of a string. For combining strings in this way, continue to use concatenation. Concatenation is the combining (adding together) of strings, as in A$+B$+C$.

Statement format:
INSERT("sub string","main string",insert position of sub-string)
or
INSERT(string variable name,string variable name,insert position of sub-string)

INSERT may only be used as a program statement. *It will not perform correctly used as a command.* The insert position of the sub-string parameter specifies the last position of the main string *before* inserting the sub-string. For example: to insert the sub-string "STATEMENT" between the two words of the main string "INVOICE NUMBER", the position for sub-string is 8; the seven letters of INVOICE plus the space following INVOICE. The sub-string "STATEMENT" will begin one position beyond the insert position number.

Either a character string or string variable may be specified in the INSERT statement. The string is limited to 255 characters. A string greater than 255 characters will cause the error message ?CREATED STRING TOO LONG. Attempting to add to the end of a main string by

specifying an insert position greater than the length of the main string will cause the error ?INSERT PARAMETER TOO LARGE.

For a demonstration of this command, if the chapter program is still in memory, which I hope it is, LIST the program and add the statement lines shown below. For those of you who have cleared memory and haven't saved the program or are reading but not keying in, the program is shown below in its entirety.

Listing of Chapter Program

```
10 PRINT "[CLEAR]":PRINT
20 A$="SIMONS' ARCADE":CENTRE A$:PRINT
30 CENTRE "1200 WILSON DRIVE":PRINT
40 CENTRE "WEST CHESTER, PA. 19380"
43 PRINT:PRINT
45 PRINT "INVOICE NUMBER 1024"
50 END
```

When adding the following lines, be sure to leave a space at the end of both character strings, "PENNY " and "STATEMENT ". The statement lines to be added are;

```
25 S$="PENNY "
44 B$=INSERT(S$,A$,8):PRINT B$
45 PRINT INSERT("STATEMENT ","INVOICE NUMBER",8)
```

RUN the program.

INST

This statement is used to *overwrite* a portion of a main string with a sub string.

Statement format:
INST("sub-string","main string",insert position of sub-string)
or
INST(string variable name,string variable name,insert position of sub-string)

INST may only be used as a program statement. *It will not perform correctly used as an immediate mode command.* Insert position is determined and specified in the same way for INST and INSERT.

Either a character string or string variable may be specified in the INST statement. The string is limited to 255 characters. A string greater than 255 characters will cause the error message ?CREATED STRING TOO LONG.

For those of you who may be reading this book but not keying in, listed below is the chapter program, as it stands now, which we are modifying for each new chapter command.

Listing of Chapter Program

```
10 PRINT "[CLEAR]":PRINT
20 A$="SIMONS' ARCADE":CENTRE A$:PRINT
25 S$="PENNY "
30 CENTRE "1200 WILSON DRIVE":PRINT
40 CENTRE "WEST CHESTER, PA. 19380"
43 PRINT:PRINT
44 B$=INSERT(S$,A$,8):PRINT B$
45 PRINT INSERT("STATEMENT ","INVOICE NUMBER",8)
50 END
```

LIST the program in memory and add the following statements for a demonstration of INST.

```
46 PRINT:PRINT
47 PRINT INST("VIDEO",B$,8)
```

The sub-string "VIDEO" will replace the characters PENNY in the character string B$. RUN the program. This new character string can first be created, then printed, enabling you to retain the new string in memory. Change statement line 47, as follows.

```
47 C$=INST("VIDEO",B$,8):PRINT C$
```

RUN the program.

Remember that the sub-string overwrites the main string. Take care that the sub-string is not larger than the portion of the main string that you wish to overwrite. For example, using INST to overwrite the characters PENNY with "COMPUTER" would also overwrite the space following PENNY and the AR of ARCADE because COMPUTER is three characters larger than PENNY. To demonstrate this, add the following statement line to the program in memory, and LIST and RUN the program again.

```
47 PRINT:PRINT INST("COMPUTER",B$,8)
```

Of course, if you are overwriting the last word in a main string, it doesn't matter if the sub string is longer than that word. Change statement line 47 as follows, and RUN the program again.

```
47 PRINT:PRINT INST("COMPUTER GAMES",B$,14)
```

PLACE

This command eliminates the need to count main string characters for the insert position when using INSERT or INST. PLACE searches a main string for the specified sub-string and returns the character position of the sub-string. If the main string does not contain the specified sub-string, a value of 0 is returned.

Statement format:
PLACE("sub-string","main string")
or
PLACE(string variable name,string variable name)

PLACE should only be used within a program. *It will not always perform correctly used as an immediate mode command.* The length of the sub-string must be less than the length of the main string. If the sub-string is longer than the main string, a value of 0 will be returned.

The position value returned by PLACE can be printed or stored as a numeric variable. This position value is the position of the first character of the sub-string found in the main string. This is one greater than used for the insert position parameter of INSERT and INST. Here are several examples:

PRINT PLACE("ACCOUNTS","ACTIVE ACCOUNTS PAYABLE") returns the value 8.

PRINT PLACE("PAYABLE","ACTIVE ACCOUNTS PAYABLE") returns the value 17.

PRINT PLACE("ACTIVE","ACTIVE ACCOUNTS PAYABLE") returns the value 1.

To demonstrate the usefulness of PLACE, add the following statement line to the program in memory, and LIST the program.

```
47 P=PLACE("PENNY",B$):PRINT INST("BASIC",B$,P-1)
:PRINT "[DOWN 2]"
```

Again, for those of you who are not keying in this program, it is listed below in its entirety—including the new line shown above.

Listing of Chapter Program

```
10 PRINT "[CLEAR]":PRINT
20 A$="SIMONS' ARCADE":CENTRE A$:PRINT
25 S$="PENNY "
30 CENTRE "1200 WILSON DRIVE":PRINT
40 CENTRE "WEST CHESTER, PA. 19380"
43 PRINT:PRINT
44 B$=INSERT(S$,A$,8):PRINT B$
45 PRINT INSERT("STATEMENT ","INVOICE NUMBER",8)
46 PRINT:PRINT
47 P=PLACE("PENNY",B$):PRINT INST("BASIC",B$,P-1):
PRINT "[DOWN 2]"
50 END
```

The PLACE statement will store the position value of PENNY in the character string SIMONS' PENNY ARCADE in the numeric variable P. The PRINT statement will use the value of P minus 1 to overwrite PENNY with the word BASIC. The value of P-1 must be used because the INST statement uses an insert position of one space less than the location position returned by PLACE.

RUN the program. By using PLACE, you did not have to know the location of the sub-string before using INST.

PLACE can also be used to find the correct insert position for using INSERT to add to a character string, as the following statement line demonstrates.

ENTER:
```
48 PP=PLACE("ARCADE",B$):PRINT
INSERT("POWER ",B$,PP-1)
```

The PLACE statement will locate the position of the word ARCADE in B$. The position value obtained will correctly locate "POWER" in front of "ARCADE" in the B$ character string, using the INSERT command. The new character string will then be displayed.

RUN the program.

DUP

This command provides an easy method for creating a new, and larger, character string by duplicating it a defined number of times.

This is particularly useful for inserting spaces into a character string or for repeating characters.

Command format:
DUP("string",duplication number)
or
DUP(string variable,duplication number)

DUP can be used as an immediate mode command or as a program statement. The string created by DUP cannot exceed 255 characters. A string longer than 255 characters will produce the error message ?CREATED STRING TOO LONG.

Following are two examples of the use of DUP, using the immediate mode. The use of DUMP following these commands will display the values of the variables created by the commands, as well as the variable values from the program in memory.

```
ENTER:   PRINT DUP("TOUCHDOWN! ",3)
         A$="ANNUAL SUPER ":B$=DUP(" ",80):C$=DUP
         ("SALE! ",3):PRINT A$;B$;C$
         DUMP
```

For those of you who are not keying in, the chapter program, as it is now in memory, is shown below.

Listing of Chapter Program

```
10 PRINT "[CLEAR]":PRINT
20 A$="SIMONS' ARCADE":CENTRE A$:PRINT
25 S$="PENNY "
30 CENTRE "1200 WILSON DRIVE":PRINT
40 CENTRE "WEST CHESTER, PA. 19380"
43 PRINT:PRINT
44 B$=INSERT(S$,A$,8):PRINT B$
45 PRINT INSERT("STATEMENT ","INVOICE NUMBER",8)
46 PRINT:PRINT
47 P=PLACE("PENNY",B$):PRINT INST("BASIC",B$,P-1):
PRINT "[DOWN 2]"
48 PP=PLACE("ARCADE",B$):PRINT INSERT("POWER ",B$,
PP-1)
50 END
```

LIST the program in memory and add the following statement lines to demonstrate DUP in the program mode. Be sure to leave four spaces at the end of the character string SIMONS'.

```
49 FOR I=1 TO 6:PRINT:NEXT I
5Ø PRINT DUP("SIMONS'    ",4)
```

These statements will create a string of SIMONS' across the bottom of the screen. RUN the program.

CSET

This command allows you to switch between the two Commodore 64 character sets—the upper case/graphic character set and the upper/lower case character set.

Command format:
CSET n

The parameter n designates one of three functions, as shown below. Only the first two functions are used with low resolution screens (40 columns by 25 rows).

Ø specifies the upper case/graphics character set.
1 specifies the upper/lower case character set.
*2 re-displays the last high-resolution or multicolor graphics screen.
* CSET 2 will be reviewed in Chapter 17.

The character set specified will affect the output on the entire screen. To alternate between character sets either: (1) the screen must first be cleared, or (2) program execution must be temporarily stopped using the PAUSE command and the characters "replaced." **Note:** You cannot use the easy Commodore BASIC method of shifting character sets—SHIFT+CMDR—when using the Simon's BASIC cartridge.

Clear your screen and enter the following commands in the immediate mode. The character string "SIMONS' BASIC" will be printed first in lower-case letters. Then the display will be changed to upper-case letters, and back to lower-case letters.

```
CSET 1:PRINT "[DOWN 5]" TAB(12) "SIMONS' BASIC":
PAUSE 3:CSET Ø:PAUSE 3:CSET 1
```

Enter CSET Ø to return the screen to the upper-case/graphic character set.

Printed below is the latest version of the chapter program in memory.

Listing of Chapter Program

```
10 PRINT "[CLEAR]":PRINT
20 A$="SIMONS' ARCADE":CENTRE A$:PRINT
25 S$="PENNY "
30 CENTRE "1200 WILSON DRIVE":PRINT
40 CENTRE "WEST CHESTER, PA. 19380"
43 PRINT:PRINT
44 B$=INSERT(S$,A$,8):PRINT B$
45 PRINT INSERT("STATEMENT ","INVOICE NUMBER",8)
46 PRINT:PRINT
47 P=PLACE("PENNY",B$):PRINT INST("BASIC",B$,P-1):
PRINT "[DOWN 2]"
48 PP=PLACE("ARCADE",B$):PRINT INSERT("POWER ",B$,
PP-1)
49 FOR I=1 TO 6:PRINT:NEXT I
50 PRINT DUP("SIMONS'    ",4)
```

LIST the program in memory, and add the following statement to produce screen output in lower-case letters.

```
22 CSET 1
```

RUN the program, then enter the command **CSET Ø** to restore the upper case character set.

INV

This command can be used in place of the RVS ON and RVS OFF keys. INV reverses the background of characters contained within a user-specified screen area. Characters currently displayed on a normal background will be displayed in a reverse field by INV, while characters currently displayed on a reverse field will be displayed normally.

Command format:
INV row,column,width,depth

The four parameters of INV designate the screen area within which the character display is to be changed. Since INV uses a low-resolution screen, rows are numbered 0 through 24 and columns are numbered 0 through 39. Width specifies the number of columns to be changed, while depth specifies the number of rows to be changed.

Clear your screen and enter the following commands in the immediate mode to display the character string "SIMONS' BASIC" in reverse field, pause 3 seconds, and change back to a normal field.

```
PRINT "[DOWN 5]" TAB(12) "SIMONS' BASIC":INV 7,12,
13,1:PAUSE 3:INV 7,12,13,1
```

If the exact position of the characters to be changed is not known, simply designate a large enough screen area in INV to include the characters to be changed.

LIST the demonstration program in memory. Add the INV statement shown below to display the character string printed in line 20 in both lower case and reverse field.

```
23 INV 0,5,30,5
```

RUN the program.

Since INV only affects what has already been printed, a large screen area can be specified by width (30) and depth (5). The exact location of "simons' arcade" did not have to be specified. To include the second line of text in INV command, change statement number 23 to 35 and remove statement number 23. RUN the program again.

To include the first three lines of print, change statement number 35 to 55 and remove statement number 45. RUN the program again. Enter **CSET 0** to change the character set back to upper-case.

The seven Simons' BASIC commands reviewed in this chapter will add a great deal of flexibility to the format and appearance of your programs.

8 Input Validation

This chapter will use a demonstration program for each of the three Simons' BASIC methods of validating data input from the keyboard—INKEY, ON KEY and RESUME, and FETCH. INKEY is used to test for and store the value of a function key input. ON KEY is used for accepting a user-specified input character from the keyboard and branching based upon that input. RESUME is a companion command for ON KEY, performing a branch back to the last ON KEY command performed. And FETCH limits the type of input and number of characters accepted. At the end of the chapter, the three input commands will be used together in a summary demonstration program.

INKEY

INKEY is a numeric variable used to hold the number of the function key input from the keyboard. This variable is the argument used for an ON GOTO or ON GOSUB statement to perform a conditional branch based on the function key input. INKEY recognizes the function keys f1 through f8, providing eight branching possibilities. Because the value of INKEY is input from the keyboard, the statement must be placed into a loop, continuing the keyboard scan until a function key is pressed. This is the same programming technique used with the GET statement. The format for the INKEY statements is shown below.

Statement format:
ON INKEY GOTO statement line list:GOTO the INKEY statement line

or

ON INKEY GOSUB statement line list: GOTO the INKEY statement line

This statement will pause a program until one of the function keys, f1 through f8, is pressed. When a function key is entered, the ON GOTO branch is made using the number of the function key pressed to choose from the statement line list.

The following program demonstrates ON INKEY GOTO, using all 8 function keys. Use f5 (if it is still programmed for AUTO 10,10), enter AUTO 10,10, or enter the line number shown in the listing below when entering this program.

Program Listing

```
10 PRINT "PRESS ANY FUNCTION KEY, F1 THROUGH F8"
20 ON INKEY GOTO 30,40,50,60,70,80,90,100:GOTO 20
30 PRINT "F";INKEY,"LINE 30":GOTO 10
40 PRINT "F";INKEY,"LINE 40":GOTO 10
50 PRINT "F";INKEY,"LINE 50":GOTO 10
60 PRINT "F";INKEY,"LINE 60":GOTO 10
70 PRINT "F";INKEY,"LINE 70":GOTO 10
80 PRINT "F";INKEY,"LINE 80":GOTO 10
90 PRINT "F";INKEY,"LINE 90":GOTO 10
100 PRINT "F";INKEY,"LINE 100":GOTO 10
110 END
```

Each time a function key is pressed (touch the key lightly, or you'll get a double response), the PRINT statement branched to will display the function key number (the value of INKEY), and the program line number being performed.

RUN the program, testing each of the eight function keys, in any order, and press the RUN/STOP key to end the program.

Clear computer memory and enter the following short program using ON GOSUB with the function keys f1, f2, and f3. Enter AUTO 10,10 for automatic line numbering, or enter the line numbers shown.

Program Listing

```
10 PRINT "PRESS F1, F2, or F3"
20 ON INKEY GOSUB 30,40,50:GOTO 20
30 PRINT "F";INKEY,"LINE 30":RETURN
40 PRINT "F";INKEY,"LINE 40":RETURN
50 PRINT "F";INKEY,"LINE 50":RETURN
```

Only function keys f1 through f3 can be input, because only three statements are specified for branching.

RUN the program, pressing a function key each time the program halts. Press the RUN/STOP key to end the program.

INKEY will be used later in the chapter in a menu-driven program.

ON KEY

This command scans the keyboard for a valid character input. The input character is "valid" if it is a character specified in the ON KEY statement. Because the value of ON KEY is input from the keyboard, the statement must be placed into a loop until a function key is pressed. Again, this is the same programming technique used with the GET statement.

Use of ON KEY requires three statements, as shown below.

Statement format:
ON KEY "characters",:GOTO line number
GOTO the ON KEYstatement line
or
ON KEY string variable,:GOTO line number
GOTO the ON KEY statement line

The ON KEY statement specifies the character(s) or string variable to be used for the keyboard scan. Note that this statement must be separated from the GOTO statement *by a comma* ***and*** *a colon*. The GOTO statement following ON KEY is used to branch out of the scanning routine. The second GOTO statement causes the continuous keyboard scan and *must appear on a separate statement line*. If this statement is entered on the same statement line the program just will not run properly. You will probably not get an error message, although I did encounter a TYPE MISMATCH error once for this error.

Although many characters may be specified in the ON KEY statement, only one of those characters can be input. Any character not specified in the ON KEY command will be ignored. When one of the specified characters is input, the GOTO statement branch following the ON KEY command is performed.

The ASCII character code number of the character input for the ON KEY command is stored in the variable ST. The value of CHR$(ST), or ST, can then be used in or by another statement, such as decision making and branching statements.

Valid ON KEY Characters

Character	Code	Character	Code	Character	Code
Space	32	6	54	J	74
!	33	7	55	K	75
"	35	8	56	L	76
$	36	9	57	M	77
%	37	:	58	N	78
&	38	;	59	O	79
'	39	<	60	P	80
(	40	=	61	Q	81
)	41	>	62	R	82
*	42	?	63	S	83
+	43	@	64	T	84
,	44	A	65	U	85
-	45	B	66	V	86
.	46	C	67	W	87
/	47	D	68	X	88
0	48	E	69	Y	89
1	49	F	70	Z	90
2	50	G	71	[	91
3	51	H	72	£	92
4	52	I	73	↑	94
5	53				

Figure 8.1: Chart showing the valid ON KEY Characters.

Valid characters are those having the character codes 32 through 94, except for 34 (quote) and 93 (right bracket), allowing 61 characters to be used. These include all alphabetic, numeric, punctuation, and mathematic symbols. A complete list of the valid characters is shown in Figure 8.1.

Although the shifted and Commodore graphic characters can be used as input also, the character code held in ST is not usable. Trying to read either ST or CHR$(ST) will cause ?ILLEGAL QUANTITY ERROR.

The following program will give a demonstration of ON KEY using the two characters A and B. Use the programmed function key (f5) or AUTO 10,10 for automatic line numbering. If you're not using this convention, enter the line numbers shown with the statement lines in the program below.

Program Listing

```
10 PRINT "PRESS KEY A OR B"
20 ON KEY "AB",:GOTO 40
30 GOTO 20
40 PRINT ST, CHR$(ST)
50 IF CHR$(ST)="A" GOTO 70
60 PRINT "YOU PRESSED B":GOTO 80
70 PRINT "YOU PRESSED A"
80 END
```

This program uses the ON KEY command to scan for keyboard input of either of the two letters A or B. It then tests the value of CHR$(ST), which contains the character input from the keyboard, for a program branch. When run, the program will pause waiting for the character A or B to be pressed.

RUN the program twice, testing A, then B, as the keyboard input. You may also want to test other characters to prove that they will not be accepted as input.

The ASC code 65 is given for ST, and the value "A" for CHR$(ST) when the key A is entered. The ASC code 66 is given for ST, and the value "B" for CHR$(ST) when the key B is entered.

RESUME

ON KEY has a companion statement—RESUME. This command causes a program to branch back to the last ON KEY command.

Statement format:
RESUME

To demonstrate this command, edit the program in memory as follows.

```
ENTER:   60 PRINT "YOU PRESSED B":RESUME
         80 RESUME
         90 END
```

After printing ST, CHR$(ST), and the message telling which key was pressed, the program will return to statement line 20 and halt until either key A or key B is pressed.

RUN the program using both A and B as input at least once. Press the RUN/STOP key to end the program.

Note: The user guide that is included with the Simons' BASIC cartridge specifies that you must use the DISABLE command to turn ON KEY off. I have found this command completely unnecessary, even when following an ON KEY routine with statements such as GET or INPUT. This does not generate the "recursive jumps" promised in the user guide if the DISABLE command is not used. Since this command is "extra baggage," it is not covered in this book. However, I do not have Simons' BASIC on tape or disk, only on cartridge. It's possible that the cartridge version was modified to avoid this problem without need for the DISABLE command. If those of you who have the tape or disk version have a problem with ON KEY recursive jumps, you should use the DISABLE command following your ON KEY routine.

FETCH

This input statement allows the programmer to limit the program user's keyboard input by controlling the type of characters and number of characters input. FETCH can be used in place of an INPUT statement. Unlike INPUT, however, FETCH does not include an input message, nor does it provide an input prompt (?). The programmer must provide the input message in a PRINT statement preceding FETCH.

Statement format:
FETCH "control character",character limit,variable name

The three FETCH control characters are shown in Figure 8.2. The variable specified for the first two control characters—HOME and CURSOR RIGHT—*must* be a string variable. The variable specified for the control character CURSOR DOWN *must* be numeric.

Control Character	Control Character Key	Valid Characters
" [S] "	HOME	Unshifted alphabetic characters only.
" [J] "	CURSOR RIGHT	Shifted and unshifted alphabetic characters only.
" [Q] "	CURSOR DOWN	Numeric characters only.

Figure 8.2

If a numeric variable is specified in the FETCH statement when using either the HOME or CURSOR RIGHT control character, the message ?REDO FROM START will be printed after characters are input. The only way to correct this error is to stop the program using the RUN/STOP and RESTORE keys, and correct the FETCH statement.

However, if a string variable is specified in FETCH when using the CURSOR DOWN control key, the statement will be performed. It will accept all non-alphabetic shifted and unshifted keys—*including space*—in addition to numeric keys, except the pound sign (£), exponential sign (↑), left and right brackets ([ ]), and comma. (A comma will be accepted but ignored, as will all characters following it, and generate the message EXTRA IGNORED.) Using this control key for entering a certain type of information, such as telephone numbers, social security numbers, and so forth, can be very useful since the space character cannot be accepted by the other two control keys.

The program shown below demonstrates all three standard uses of FETCH, plus the deliberate use of a character string for numeric input. The command illustrated is shown to the left of the program listing. To save keying in time, use automatic line numbering by entering AUTO 10,10 or a programmed function key. If you use automatic line numbering, be sure not to enter the line numbers yourself!

Command Illustrated	Program Listing
CENTRE	10 PRINT "[CLEAR]":PRINT:CENTRE "FETCH TEST": PRINT:PRINT
FETCH	20 PRINT "YOUR FIRST NAME?":FETCH"[HOME]",8,N$
FETCH	30 PRINT "YOUR ZIP CODE?":FETCH "[CURSOR DOWN]",5,Z
FETCH	40 PRINT "YOUR TELEPHONE NUMBER?":FETCH "[CURSOR DOWN]",8,T$
FETCH	50 PRINT "YOUR SYMBOL?":FETCH "[CURSOR RIGHT]",4,S$
	60 PRINT N$,,Z,T$,S$
	70 END

After each message prints, the cursor will remain on the screen waiting for your input. Type your input and press the RETURN key. The input will then be cleared from the screen and the next PRINT statement performed. After all FETCH input is entered, the FETCH variable values will be printed.

The first input will be your first name. Two FETCH statements would be needed for first and last name, three for first, last, and middle name or initial because the space character will not be accepted by either control character used for alphabetic characters. The last input—YOUR SYMBOL—is to demonstrate the use of the graphic keys. Choose any four shifted or Commodore graphic keys for this entry.

RUN the program, and attempt to enter the incorrect type of input—alphabetic in numeric or numeric in alphabetic—before entering the correct data.

To keep the FETCH input from being erased from the screen when the next statement is displayed, add a PRINT statement following each FETCH statement. To do this, either edit statement lines 20, 30, 40, and 50, or

```
ENTER:   25 PRINT
         35 PRINT
         45 PRINT
         55 PRINT
```

RUN the program.

The following menu-driven summary program will demonstrate each

of the three commands covered in this chapter. This type of program is an excellent place to make use of INKEY, although a menu-driven program would usually have more program options. ON KEY is used in a subroutine to test whether the user wishes to continue or end the program. The FETCH statement is used throughout the program, in place of the INPUT statement. Each Simons' BASIC command illustrated is shown to the left of the program listing.

You may use AUTO 10,10 to enter the following program.

Command Illustrated	Program Listing
CENTRE	10 PRINT "[CLEAR]":PRINT:CENTRE "TRIVIA BY TOPIC":PRINT:PRINT
	20 PRINT:PRINT "PRESS THE FUNCTION KEY SHOWN": PRINT "TO CHOOSE YOUR TOPIC."
	30 PRINT:PRINT "F1=U.S. SPACE FLIGHT": PRINT "F2=SPORTS"
INKEY	40 ON INKEY GOTO 50,240:GOTO 40
CENTRE	50 PRINT "[CLEAR]":PRINT:CENTRE "U.S. SPACE FLIGHT TRIVIA":PRINT
	60 PRINT:PRINT "LAST NAME OF THE FIRST
FETCH	AMERICAN IN SPACE?":FETCH "[HOME]",7,S$:PRINT
	70 IF S$="SHEPARD" THEN PRINT "RIGHT!": GOTO 90
	80 PRINT "WRONG. THE ANSWER IS: SHEPARD."
	90 PRINT:PRINT "LAST NAME OF THE FIRST AMERICAN TO ORBIT THE EARTH?"
FETCH	100 FETCH "[HOME]",5,E$:PRINT
	110 IF E$="GLENN" THEN PRINT "RIGHT!": GOTO 130
	120 PRINT "WRONG. THE ANSWER IS: GLENN."
	130 PRINT:PRINT "LAST NAME OF THE FIRST AMERICAN TO WALK IN SPACE?"
FETCH	140 FETCH "[HOME]",5,W$:PRINT
	150 IF W$="WHITE" THEN PRINT "RIGHT!": GOTO 170
	160 PRINT "WRONG. THE ANSWER IS: WHITE."
	170 PRINT:PRINT "LAST NAME OF THE FIRST MAN TO WALK ON THE MOON?"
FETCH	180 FETCH "[HOME]",9,M$:PRINT

```
                190 IF M$="ARMSTRONG" THEN PRINT
                "RIGHT!":GOTO 210
                200 PRINT "WRONG. THE ANSWER IS: ARMSTRONG."
CENTRE          210 PRINT:PRINT:CENTRE "END OF SCIENCE
                TRIVIA":PRINT
                220 GOSUB 460
                230 PRINT "[CLEAR]":GOTO 20
CENTRE          240 PRINT "[CLEAR]":PRINT:CENTRE "SPORTS
                TRIVIA":PRINT
                250 PRINT:PRINT "HOW MANY LAPS USUALLY MAKE
                UP THE":PRINT "INDY 500?"
FETCH           260 FETCH "[CURSOR DOWN]",3,L:PRINT
                270 IF L=200 THEN PRINT "RIGHT!":GOTO 290
                280 PRINT "WRONG. THE ANSWER IS: 200."
                290 PRINT:PRINT "IN BASKETBALL, HOW MANY FEET
                IS THE NET ABOVE THE COURT?"
FETCH           300 FETCH "[CURSOR DOWN]",2,N:PRINT
                310 IF N=10 THEN PRINT "RIGHT!":GOTO 330
                320 PRINT "WRONG. THE ANSWER IS: 10."
                330 PRINT:PRINT "HOW MANY PLAYERS ARE ON AN
                OUTDOOR POLO TEAM?"
FETCH           340 FETCH "[CURSOR DOWN]",1,P:PRINT
                350 IF P=4 THEN PRINT "RIGHT!":GOTO 370
                360 PRINT "WRONG. THE ANSWER IS: 4."
                370 PRINT:PRINT "HOW MANY INCHES ACROSS DOES
                AN"
                380 PRINT "ICE HOCKEY PUCK MEASURE?"
FETCH           390 FETCH "[CURSOR DOWN]",1,H:PRINT
                400 IF H=3 THEN PRINT "RIGHT!":GOTO 420
                410 PRINT "WRONG. THE ANSWER IS: 3."
CENTRE          420 PRINT:PRINT:CENTRE "END OF SPORTS
                TRIVIA":PRINT
                430 GOSUB 460
                440 PRINT "[CLEAR]":GOTO 20
                450 END
                460 PRINT:PRINT "PRESS C TO CHOOSE ANOTHER
                TOPIC OR E TO END THE PROGRAM."
ON KEY          470 ON KEY "CE",:GOTO 490
                480 GOTO 470
                490 IF CHR$(ST)="E" GOTO 450
                500 RETURN
```

RUN the program.

Now you can easily write your own menu-driven programs, and create more sophisticated programs with the help of these new input statements.

9 Output Options

This chapter reviews three formatting commands—PRINT AT, USE, and MOVE—for screen output, a new listing command—OPTION, and a command for hard copy printer output—HRDCPY. These commands are useful in all types of programming; for business, education, and entertainment. Again, this chapter will use a demonstration program that will be added to as each of these commands is reviewed. If you do not complete this chapter at one time, SAVE the program before you stop. Then LOAD the program before you continue in the chapter.

AT

This PRINT function positions text at a user-specified screen location. Since it uses column and row coordinates, it is a great deal more powerful that either TAB or SPC.

Function format:
AT(column,row)"character string"
or
AT(column,row)string variable name

The number of AT functions specified in a PRINT command is only limited by the length of a PRINT line—two screen lines. AT may use either a character string or string variable. There may be no space between AT and the left parenthesis of the column and row coordinate parameter. A space will cause a ?BAD SUBSCRIPT ERROR.

The following program will demonstrate this function and review several of the commands from previous chapters. The command illustrated is shown to the left of the statement line in which it appears. To use automatic line numbering for the program, use AUTO 10,10. Or,

use a programmed function key—f5 if you still haven't turned off your Commodore since Chapter 5 or you saved and reloaded the function key assignments! If this is the case, your dedication to Simons' BASIC is admirable! Those of you who are not convinced to change your old habits and save a little keyboarding time will enter the statement line numbers shown.

```
                NORTHWEST INTERNISTS, P.C.
                   JAMES S. BOND, M.D.
                   KEITH E. POUND, M.D.
                 717 NORTH BROAD STREET
                 NEW YORK, NEW YORK 10021

INTERNAL MEDICINE                        MU-8-0400

-------------------------------------------------------

                                         PAY-
DATE          CODE   CHARGES    MENTS   BALANCE

-------------------------------------------------------


READY.
□
```

Figure 9.1: Northwest Internists Billing Form.

Command Illustrated	Program Listing
	10 PRINT"[CLEAR]":PRINT
CENTRE	20 CENTRE "NORTHWEST INTERNISTS, P.C." :PRINT
CENTRE	30 CENTRE "JAMES S. BOND, M.D.":PRINT
CENTRE	40 CENTRE "KEITH E. POUND, M.D.":PRINT
CENTRE	50 CENTRE "717 NORTH BROAD STREET":PRINT
CENTRE	60 CENTRE "NEW YORK, NEW YORK 10021" :PRINT

```
AT          70 I$="INTERNAL MEDICINE":PRINT AT(0,8)
            I$ AT(30,8)"MU-8-0400"
DUP         80 PRINT:PRINT DUP("--",40):PRINT:PRINT
AT          90 PRINT AT(25,11)"PAY-"
AT          100 PRINT AT(0,12)"DATE" AT(10,12)"CODE"
            AT(16,12)"CHARGES" AT(25,12)"MENTS";
AT          110 PRINT AT(32,12)"BALANCE"
DUP         120 PRINT:PRINT DUP("--",40):PRINT:PRINT
            130 END
```

The screen output will be a billing form used by a medical practice partnership.

RUN the program. Note the ease of formatting this bill, and the handy use of DUP for drawing lines. The screen output is shown in Figure 9.1.

USE

As mentioned in Chapter 1, this is the Simons' BASIC command used to format lists of numbers—align the decimal points. Because the decimal point symbol is a character, numbers formatted with USE are characters, either part of a character string or a string variable.

Command format:
USE "#.#","character string" *or* string variable name
or
USE "text#.#text","character string" *or* string variable name

Hash marks are used to show the format image to be used by the character string or string variable. The decimal must be displayed in the character string. Only one string or string variable may be specified in a USE command. If more than one is specified it will cause ?SYNTAX ERROR. Only one decimal may be used in the format image for the USE statement to execute correctly.

Text may also be a part of the format image, i.e., "$###,###.##". To illustrate this example,

ENTER: `A$="44444.44":USE "$###,###.##",A$`

Your output is: $ 44,444.44. In the immediate mode, the character string for USE must be given before the USE command is entered. This

is not necessary when USE is a program statement. In the above example, the dollar sign and the comma are text and are displayed by the format image when the character string is printed.

The following statement lines will add billing data to the program in memory, demonstrating the benefits of USE. LIST the program, and if you wish to use automatic line numbering enter AUTO 13Ø,1Ø. Then enter the following program lines. The command illustrated is shown to the left of the program statement.

Command Illustrated	Statements to be added:
	13Ø READ D$,CO$,CH$,P$,B$:IF D$="-99" GOTO 19Ø
	14Ø PRINT D$ TAB(1Ø) CO$;
USE	15Ø PRINT TAB(16):USE "$###",CH$
USE	16Ø PRINT TAB(25):USE "$###",P$
USE	17Ø PRINT TAB(32):USE "$###",B$:PRINT
	18Ø GOTO 13Ø
	19Ø PRINT:PRINT:PRINT "PLEASE PAY LAST AMOUNT IN BALANCE COLUMN"
	2ØØ DATA 1/12/85,EKG,13Ø,6Ø,7Ø, 2/Ø1/85, OV,25,25,7Ø,3/12/85,LAB,6Ø,6Ø,7Ø
	21Ø DATA 3/12/85,OV,25,5Ø,45
	22Ø DATA -99,-99,-99,-99,-99
	23Ø END

The TAB function is used to align the formatted character strings (USE) with the column headings, using the column parameter of each AT function (statement line 1ØØ).

RUN the program.

MOVE

This command is used to duplicate a specified portion of the screen and display the duplicate at a given screen location. The exact location of the portion of the screen to be copied and the location that the duplicate is to occupy must be specified in the MOVE command.

This command requires six parameters; the first four specify the portion of the screen to be copied, and the last two give the location of the duplicate. The first two parameters specify the row and column where

the information to be duplicated begins. The third parameter gives the width (screen length occupied) of the information to be copied, and the fourth parameter tells the number of lines it occupies. The fifth parameter specifies the receiving row of the duplicate; and the sixth parameter gives the column in which the duplicate will begin. All six parameters must be used. In setting these parameters, remember that this screen type—the low-resolution, normal screen—is configured 25 rows, numbered 0 through 24, by 40 columns, numbered 0 through 39.

Command format:
MOVE row,column,width,lines,receiving row,receiving column

The width of the area to be duplicated plus the receiving column cannot exceed 40, the screen width. Likewise, the number of lines (depth) of the area to be duplicated plus the receiving row cannot exceed 25, the screen height. If either of these conditions occur, it will cause the error condition ?BAD MODE.

In order to use MOVE in the immediate mode, the characters to be copied must reside in memory before MOVE is entered. This can be accomplished by storing the data of the area to be duplicated in a variable before entering the MOVE command, as shown in the examples below. Because the screen is cleared first, the data will be printed on row 0, beginning in column 0.

ENTER:
```
A=1234:PRINT "[CLEAR]";A:MOVE 0,0,5,1,
24,35:PAUSE 10
```

In this example, the numbers 1234 will first be printed in the upper lefthand corner of your screen, on row 0—the first screen row—beginning in column 0, covering 5 print positions on 1 line. MOVE will duplicate this value in the lower righthand corner of your screen, on row 24—the last screen row—beginning in column 35. This will cause the last digit of the data to be printed in column 39—the last print position on a row. Note that the leading blank must be included in the width for the entire number to print. Using 4 for the width parameter of MOVE would only print the first three digits—123. The ten second PAUSE allows you to review the screen. **Note**: READY always prints on the screen line following the normal screen output, ignoring the MOVE display.

```
ENTER:    A$="THE MICRO FACTORY":PRINT "[CLEAR]";
          A$:MOVE Ø,Ø,17,1,12,12:PAUSE 1Ø
```

This example causes a character string printed at the first screen location (Ø,Ø) to be duplicated at a new screen location. That location is on row 12, beginning at column 12, and residing in 17 print positions.

To demonstrate MOVE in program mode, add the following statement lines to the program in memory.

```
23Ø MOVE 2,1Ø,2Ø,1,24,2Ø
24Ø GOTO 24Ø
```

LIST the program. MOVE will duplicate the 20 characters "KEITH E. POUND, M.D." located on row 2 beginning at column 10 at columns 20 through 39 on row 24. Line 24Ø will keep the screen from scrolling, so that the entire bill will stay on the screen for your review.

RUN the program, then end it by pressing the RUN/STOP key. MOVE duplicated the name of the sending doctor at the bottom, right-hand corner of the screen, copied from the heading.

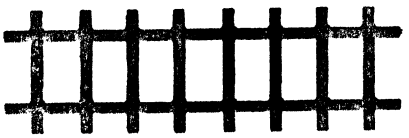
If you have a letterhead or newsletter header containing a graphic, you can "move" it to another part of the page, as shown in the short program below. This program will be added to the front of the program in memory and end with a GOTO loop.

Use automatic line numbering by entering AUTO 1,1 or key in the line numbers shown in the program listing. (**Note**: In lines 2 and 3, SHIFT+1Ø means to hold down the SHIFT key and press the + key ten times.)

Program Listing

```
1 PRINT "[CLEAR]":PRINT
2 PRINT "[SPACE2, SHIFT +1Ø]"
3 PRINT "[SPACE2, SHIFT +1Ø] RIDING THE RAILS!"
4 PRINT:PRINT:CENTRE "NEWSLETTER":PRINT
5 PRINT:PRINT "THIS MONTH'S RAILROAD NEWS INCLUDES
MANY PRICE-BREAKING FARES."
6 PRINT:PRINT TAB(5) "NEW YORK TO L.A. ONLY",
"$XXX.XX"
7 MOVE 2,3,1Ø,2,23,3Ø
8 PRINT AT(Ø,24) "SEE YOU ON THE TRACKS!"
9 GOTO 9
```

RUN the program. The screen output is shown in Figure 9.2.



RIDING THE RAILS!

NEWSLETTER

THIS MONTH'S RAILROAD NEWS INCLUDES MANY PRICE-BREAKING FARES.

NEW YORK TO L.A. ONLY $XXX.XX

SEE YOU ON THE TRACKS!

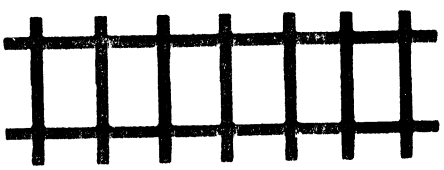


Figure 9.2: Railroaders' Newsletter

This shell newsletter may give you some ideas for your own writing projects!

Before going on to the next command, **remove** this Railroaders' Newsletter program—**statement lines 1 through 9**—leaving the Doctor bill program in memory.

OPTION

This command is used to highlight all Simons' BASIC commands when a program is listed on the screen or by the printer. This is accomplished by printing the Simons' BASIC commands in reverse-field, as though you had used reverse on. Of course, printer listings using OPTION will wear out your printer ribbon quite quickly.

Command format:
OPTION 1Ø

To demonstrate the highlighted listing using the Doctor bill program in memory:

```
ENTER:   OPTION 1Ø
         LIST
```

and use the SHIFT key to slow listing speed.

Each Simons' BASIC command, in the Doctor bill program, is printed in reverse field. CENTRE is highlighted on lines 2Ø through 6Ø, AT on lines 7Ø and 9Ø through 11Ø, DUP on lines 8Ø and 12Ø, USE on lines 15Ø through 17Ø, and MOVE on line 23Ø.

Any number between Ø and 255, except 1Ø, can be used as the OPTION parameter to turn off highlighting.

```
ENTER:   OPTION Ø
         LIST
```

Now highlighting is turned off, and the program listed in the normal manner.

HRDCPY

This is the statement used to print hard copy of a low-resolution screen (the normal screen-type, being used now) on a serial printer.

Statement format:
HRDCPY

If you have a serial printer, edit the following statement line of the billing program in memory for a demonstration of this command.

```
24Ø HRDCPY:END
```

The program output will first be displayed on the screen, and then printed on the serial printer.

These are all of the Simons' BASIC output options usable on a low-

resolution screen. High-resolution graphics output is covered in Chapter 17.

The output options covered in this chapter will make your programs more visually interesting. And, of course, this makes programming more rewarding for you.

III More Programming Power

Introduction

Following is a brief overview, by chapter, of the twelve Simons' BASIC commands that increase programming efficiency, and provide alternative methods of handling programming problems. All of these command words are reserved words, and cannot be used in variable names.

Chapter 10

The IF THEN ELSE statement tests both consequences of the IF condition, true and false, and addresses both on the same statement line. This eliminates the need for a second IF THEN statement.

The RCOMP ELSE statement is used to re-execute the last IF condition test, using different THEN and ELSE statements.

The REPEAT and UNTIL statements perform a program loop much the same as the BASIC FOR NEXT loop. However, the UNTIL statement provides a user-specified condition test at the end of the loop, making the REPEAT UNTIL loop a FOR NEXT loop with a variable ending.

Another looping option is provided by the set of statements LOOP, EXIT IF, and END LOOP. Any number of EXIT IF statements, each specifiying a condition for exiting the loop, may be used. This provides multiple condition-testing within the loop.

CGOTO is a computed GOTO statement, using the result of the com-

mand's mathematical expression for branching to a variable line number.

Chapter 11

The RESET statement gives the programmer the ability to READ a specific portion of data by branching directly to a specific DATA statement line.

The LOCAL and GLOBAL statements provide a way of reusing the same variable names within a program. When variables are reused, their values are copied and stored and, when needed, can be reset.

Chapter 12

The following four procedure statements help to change Commodore BASIC into a structured programming language.

PROC is the statement used to name a subroutine or portion of a program, making it a procedure.

END PROC is the end statement for a closed procedure, functioning the same as the RETURN statement used with GOSUB in BASIC.

CALL is the procedure version of the BASIC GOTO statement. It is used for branching—by procedure name—to an open-ended procedure.

EXEC is the procedure version of the BASIC GOSUB statement used for branching—by procedure name—to a closed procedure.

The MERGE command is used to save programming time by adding a program from a storage device to the end of the program currently in memory.

Key to Entering Program Listings

No additional key conventions are used in this section.

10 Condition Testing and Controlling Loops

Of the five Simons' BASIC statements covered in this chapter, the IF THEN ELSE statement is the one you will use most frequently. Those of you who have worked with other versions of BASIC have probably encountered the IF THEN ELSE statement and regretted not having this ability on the Commodore 64. With ELSE added to the IF THEN statement, both consequences of the IF condition can be addressed on the same statement line. Because of the two-line limitation of a Commodore BASIC statement, the creative programmer will welcome the RCOMP ELSE statement. This statement is used to re-execute the conditional test of the last IF THEN ELSE using new THEN and ELSE statements.

The FOR NEXT loop gets a big brother called REPEAT UNTIL. This pair of statements perform a program loop until a user-specified condition is met at the end of the loop—a FOR NEXT loop with a variable ending. If a condition test at the end of the loop isn't sufficient, you can use the set of statements LOOP, EXIT IF, and END LOOP. EXIT IF can be used any number of times for condition testing for exiting the loop from anywhere within the loop.

The last statement covered is the computed GOTO—CGOTO—that causes a branch to a variable line number. The line number branched to is determined by the result of the CGOTO computation.

Although you will be using some of these statements infrequently, each statement will be very useful on some occasions.

IF THEN ELSE

The IF THEN portion of this statement remains the same as in Commodore BASIC, testing the IF condition and performing the THEN statements if the condition is *true*. ELSE specifies—on the same statement line—the statements to be executed if the condition is *false*, eliminating the need for another IF THEN statement.

Statement format:
IF condition THEN statement(s):ELSE:statement(s)

GOTO can be used with or in place of THEN in this statement, the same as with IF THEN in Commodore BASIC. Note that there must be a colon *both before and after the ELSE command*. If either colon is omitted, it will cause ?SYNTAX ERROR. The colon following the ELSE command must immediately follow it; there can be no intervening spaces! The colon preceding the ELSE command may have intervening spaces.

The following program demonstrates this statement. Use AUTO 10,10, either by entering it directly or by using a programmed function key, or enter the line numbers as shown.

```
10 INPUT A
20 IF A=1 THEN B=1:ELSE:B=5
30 PRINT A,B
40 GOTO 10
```

The IF THEN ELSE statement will execute as follows. If 1 is the value input for A, both A and B will be set to the value of 1. If any number other than 1 is input as the value of A, A will have the value input and B will be set equal to 5. RUN the program, entering first 1, then any number but 1, and press the RUN/STOP and RESTORE keys to end the program.

Of course, the IF condition may use more than one condition, using a logical operator (AND, OR, NOT), and multiple THEN and ELSE statements. As always, the statement must not exceed two print lines.

Clear computer memory, enter AUTO 10,10 directly or with a function key, and the following program. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
CENTRE	10 PRINT:CENTRE "SOLVE THE PROBLEM":PRINT "[DOWN2]"
	20 A=360-60
	30 INPUT "ANSWER TO 360-60";ANS
IF THEN ELSE	40 IF A=ANS THEN PRINT "CORRECT":ELSE:PRINT "INCORRECT. TRY AGAIN!":PRINT
IF GOTO ELSE GOTO	50 IF A=ANS GOTO 60:ELSE:GOTO 30
	60 END

Note that FETCH cannot be used in place of INPUT in this program because numeric input from FETCH cannot be used for performing mathematics.

RUN the program, first using an incorrect answer then the correct answer. Do not clear computer memory. This program will be edited to use RCOMP ELSE.

RCOMP ELSE

RCOMP ELSE reuses ***only*** *the IF condition test* of the most recently executed IF THEN ELSE statement. RCOMP ELSE specifies both the THEN and ELSE statement(s) to be used with the previous IF condition test.

Statement format
RCOMP:statement(s):ELSE:statement(s)

RCOMP, which replaces BASIC's THEN, **must** be followed by a colon, and ELSE **must** be preceded by and immediately followed by a colon. If any of the three colons are omitted or there is an intervening space between ELSE and the colon a SYNTAX ERROR will be given.

Make the following change to the program in memory to demonstrate RCOMP ELSE.

ENTER: `50 RCOMP:GOTO 60:ELSE:GOTO 30`

RUN the program. The program output is the same with RCOMP

ELSE as it was with the second IF THEN ELSE statement. RCOMP is used in place of IF A=ANS. In this instance, not much keyboarding time was saved. When the IF condition test is more complex, RCOMP will save more keyboarding time. RCOMP ELSE is very useful in adding statements to the previous IF THEN ELSE statement line, as shown in the example in your *Simons' BASIC User Guide*. It is also useful in programs with complex condition testing with many resulting branches from each set of conditions.

REPEAT UNTIL

The REPEAT and UNTIL commands perform a function similar to the FOR and NEXT commands. However, REPEAT UNTIL has the additional power of controlling the loop by a condition test specified at the end of the loop. Unlike the FOR NEXT loop, the number of times a loop is executed is not specified at the beginning of the loop, nor is the lower limit or increment. Both the lower limit and increment are specified within the loop itself. When the condition test for ending the loop is met, the program proceeds to the statement following UNTIL.

Command format:
REPEAT:loop statement(s):UNTIL condition

To demonstrate this command:

ENTER: `REPEAT:A=A+1:PRINT A:UNTIL A=10`

The comparable FOR NEXT statements for the above statement would be: FOR A=1 TO 10:PRINT A:NEXT A. REPEAT replaces FOR, A=A+1 specifies the lower limit *and* increment, replacing A=1 in the FOR statement and the default step of 1, and UNTIL A=10 specifies the upper limit, replacing TO 10 in the FOR statement.

Like the FOR NEXT loop, REPEAT UNTIL loops may be nested. Nine nested REPEAT UNTIL loops is the limit. If this limit is exceeded, it will cause the error STACK TOO LARGE. When nesting REPEAT UNTIL loops, each UNTIL statement must have a matching REPEAT statement. Omitting the REPEAT statement will cause an UNTIL WITHOUT REPEAT error specifying the line number in which it occurred.

For a demonstration of nested REPEAT UNTIL loops,

ENTER:
```
REPEAT:K=K+1:REPEAT:L=L+1:PRINT K,L:UNTIL
L=3: L=Ø:UNTIL K=5
```

As you know, in a FOR NEXT loop the lower limit is reset each time the FOR statement is executed. In REPEAT UNTIL nested loops, the lower limit of the inside loop (L in the above example) must be reset after the loop is completed (L=Ø, above), and before the loop is performed again.

An important use for REPEAT and UNTIL is for READing DATA. REPEAT UNTIL does not have the controlled upper limit of the FOR NEXT loop, and avoids the use of a DATA statement "flag value" and an IF condition test. The UNTIL condition test is the "flag value."

Clear computer memory, and enter the following program for a demonstration of REPEAT UNTIL used for reading data.

ENTER:
```
1Ø REPEAT:READ A,B:PRINT A,B:UNTIL B=12
2Ø DATA 1,2,3,4,5,6,7,8,9,1Ø,11,12
3Ø END
```

RUN the program. The UNTIL condition must specify **only** the last value in the READ statement list. Otherwise, the final READ will be incomplete, causing an OUT OF DATA ERROR. If the data contained duplicates of the end value, a DATA statement "flag value", such as -99, would be needed as the value of the UNTIL condition.

An additional application to keep in mind for REPEAT UNTIL is the "delay until key is pressed procedure," as in: **REPEAT:GET K$: UNTIL K$ < > ""**.

LOOP, EXIT IF, and END LOOP

LOOP and END LOOP specify the beginning and end of the loop respectively. EXIT IF specifies the condition for exiting the loop, and may be used any number of times with the loop. When any EXIT IF condition is met, the program branches to the statement following END LOOP.

Loop format:
LOOP:loop statement(s):EXIT IF condition:END LOOP

This loop performs the same as the REPEAT UNTIL loop, except that

condition testing for exiting the loop can be performed at any points within the loop, rather than only at the end of the loop. As with the REPEAT UNTIL loop, the lower limit and increment are specified within the loop.

The following program demonstrates this loop using two EXIT IF statements within the loop. The first EXIT IF statement tests the value of the INPUT variable, I, and is executed if the value is 3; that is, the loop is then exited. The second EXIT IF statement is executed if the value of I is greater than 99. Clear computer memory, and enter the program using AUTO 10,10 or the line numbers shown.

Program Listing

```
10 J=7
20 INPUT I
30 LOOP:EXIT IF I=3:PRINT I:I=I+J:EXIT IF I>99:
END LOOP
40 PRINT:PRINT "OUT OF LOOP, I=";I
50 END
```

RUN the program at least three times using as input the number 3, a number less than 99, and a number greater than 99. The program output, using input values of 3, 84, and122, is shown in Figure 10.1.

Program Output

```
? 3
 3
OUT OF LOOP, I= 3

? 84
 84
 91
 98

OUT OF LOOP, I= 105

? 122
 122

OUT OF LOOP, I= 129
```

Figure 10.1

As the Figure 10.1 output shows, the loop was exited at the first EXIT IF statement when the value of 3 was used as input. Using 84 as the input value the loop was exited when the value of I reached a value greater than 99. When a value greater than 99 was used as input the loop was exited immediately.

Following is a demonstration program for nested loops using LOOP, EXIT IF, and END LOOP. Each loop contains one EXIT IF statement. Clear computer memory, and enter the program using AUTO 10,10 or the line numbers shown.

Program Listing

```
10 PRINT "OUTSIDE LOOP","INSIDE LOOP"
20 LOOP
30 I=I+1:PRINT I;:J=0
40 LOOP
50 J=J+1:PRINT ,,J
60 EXIT IF J=3
70 END LOOP
80 EXIT IF I=4
90 END LOOP
```

The inside loop is performed in statement lines 40 through 70. The outside loop begins on line 20 and ends on line 90. As with the REPEAT UNTIL loop, the inside loop variable must be reset (line 30, J=0) before re-executing the inside loop.

RUN the program. The output will show the performance of both loops.

CGOTO

This statement is used to branch to the line number obtained as a result of the calculation performed in the CGOTO parameter, numeric expression.

Statement format:
CGOTO numeric expression

The mathematical expression can be a number (constant), or a numeric variable, or a combination of constants, variables, and operands. For example: N*X is a numeric expression consisting of a numeric variable, an operand, and a second numeric variable. If the value of N is 20 and the value of X is 5, CGOTO will branch to line 100, N*X. In the numeric expression I*10, consisting of a numeric variable, an operand, and a constant, if the value of I is 8, CGOTO will branch to line 80, I*10.

Note: This is important for Commodore users who normally use the short hand version for entering BASIC. You **cannot** use the abbreviation of the BASIC keyword GOTO—G[SHIFT]O—when entering the

keyword CGOTO. It will cause a SYNTAX ERROR. You can use all of the abbreviations of BASIC keywords if they stand alone, but you cannot use them as part of a new Simons' BASIC keyword.

The following demonstration program is an example of CGOTO branching. All Simons' BASIC commands illustrated in this program are shown to the left of the program listing. Clear computer memory, use AUTO10,10 if you wish automatic line numbering, and enter the following program.

Command Illustrated	Program Listing
	10 PRINT:PRINT "ENTER 4, 5, OR 6. THE NUMBER 6 WILL END THE PROGRAM."
FETCH	20 FETCH "[DOWN]",1,N:PRINT
CGOTO	30 CGOTO N*10
	40 PRINT "NUMBER 4 CAUSES A BRANCH TO LINE NUMBER 40.":GOTO 10
	50 PRINT "NUMBER 5 CAUSES A BRANCH TO LINE NUMBER 50.":GOTO 10
	60 PRINT "NUMBER 6 CAUSES A BRANCH TO LINE NUMBER 60.":END

RUN the program using each of the three numbers 4, 5, and 6 as input.

The next program demonstrates the usefulness of both CGOTO and IF THEN ELSE. It would be a useful program for traffic police. The speed limit and timed speed are entered and out comes the speeders net loss, both in dollars and points. Think of how they could speed-up processing speeders at a speed trap!

Clear computer memory, enter AUTO 5,5 for automatic line numbers, and enter the following program.

Command Illustrated	Program Listing
CENTRE	5 PRINT "[CLEAR DOWN]":CENTRE "THE SPEEDER'S LOSSES":PRINT "[DOWN2]"
	10 INPUT "TIMED SPEED,POSTED SPEED";TS,PS:OS=TS-PS
IF THEN ELSE	15 IF OS>15 THEN G=6:ELSE:G=5
CGOTO	20 CGOTO G*5

IF THEN ELSE

```
25 IF OS>10 THEN F=7*(OS-10)+40:
GOTO 35:ELSE: F=4*OS:GOTO 35
30 F=(OS-15)*10+75
35 PRINT:PRINT "TIMED","POSTED":PRINT
"SPEED", "SPEED","FINE","COST"
40 C=15:PRINT TS,PS,F,C
45 PRINT:PRINT "TOTAL DUE $";F+C
50 IF OS>=20 THEN P=4:ELSE:P=3
55 PRINT "[DOWN]POINTS =";P
60 END
```

In this program speeders are fined at the rate of $4.00 per mile for up to ten miles over the speed limit, $7.00 per mile for every mile thereafter up to 15 miles over the speed limit, and $10.00 per mile for every mile more than 15 mph over the speed limit. Three points are assigned for less than 20 mph over the limit, and four points for higher speeds. The variables used in the program are:

TS = timed speed
PS = posted speed
OS = speed over the limit
G = grade of speeder (less than or greater than 15 MPH over the limit)
F = fine for speeding
C = ost of processing ticket
P = points assigned

RUN the program, using your own input for timed speed and posted speed. If you were to use 36,25 as input, your screen would look like Figure 10.2 below. If you were to use 85,55 as input, your screen would look like Figure 10.3 below.

This ends our work with Simon's BASIC condition testing and branching statements. Though you will not find a use for all of them in each of your programs, you will use all of them occasionally and some frequently.

```
                    THE SPEEDER'S LOSSES


TIMED SPEED,POSTED SPEED? 36,25

TIMED           POSTED
SPEED           SPEED          FINE          COST
 36              25             47            15

TOTAL DUE $ 62

POINTS = 3

READY.
```

Figure 10.2: "The Speeder's Losses", using 36,25.

```
                    THE SPEEDER'S LOSSES


TIMED SPEED,POSTED SPEED? 85,55

TIMED           POSTED
SPEED           SPEED          FINE          COST
 85              55             225           15

TOTAL DUE $ 240

POINTS = 4

READY.
```

Figure 10.3: "The Speeder's Losses", using 85,55.

11 Data Handling

This chapter reviews two methods of handling data within a program: (1) resetting data pointers for use by the READ statement, and (2) clearing, reusing, and restoring variable values within a program. RESET specifies the starting DATA statement line to be read, thus eliminating the need to read data items sequentially beginning with the first DATA statement. By using LOCAL and GLOBAL, variable names may be reused within a program. The value of each of the variables specified is cleared and stored, allowing reuse of the variable names. When needed, the previous value can be restored. LOCAL and GLOBAL are useful for reducing the number of variable names used within a program when a program contains a large number of variables.

RESET

As you know, in Commodore BASIC the READ statement obtains data values sequentially, beginning with the first value in the lowest numbered DATA statement. RESET gives the programmer the ability to READ any specific portion of data by branching directly to a specified DATA statement line. The last item read is controlled in the same way whether RESET is used or not, that is the upper limit of a FOR NEXT loop, a flag value, an UNTIL condition or an EXIT IF condition is met, or end of data.

Statement format:
RESET data statement line number

Only one statement line may be specified in the RESET statement. The number of data values read is controlled by other programming

statements, as noted previously. The RESET statement only specifies the line number from which the first data value will be read.

Following are two demonstration programs using this command. The first program uses an IF THEN statement with a flag value ending each DATA statement. The second program, an edited version of the first program, uses a FOR NEXT loop to control the number of data items read. Both programs use the Commodore BASIC RESTORE statement to reset the data pointer to the first data item. To enter the programs use AUTO 10,10, or enter the line numbers as shown below in the program listing. Each command illustrated is shown to the left of the statement in which it occurs.

Command Illustrated	Program Listing
	`10 INPUT "1, 2, OR 3";A`
`RESET`	`20 IF A=2 THEN RESET 90`
`RESET`	`30 IF A=3 THEN RESET 100`
	`40 IF A<1 OR A>3 GOTO 110`
	`50 READ B:IF B=-99 GOTO 110`
	`60 PRINT B`
	`70 GOTO 50`
	`80 DATA 1,2,3,-99`
	`90 DATA 4,5,6,-99`
	`100 DATA 7,8,9,-99`
	`110 RESTORE:GOTO 10`

This program does not need to check for an input value of 1, since the READ statement will automatically read the first three data items (line 80). Without a RESET statement for the value of 1, however, the RESTORE statement (line 110) is necessary. Without either a RESET or RESTORE statement, the data in line 80 cannot be read more than once. If an input value less than 1 or greater than 3 is entered, the data pointer is restored to the first data item and input is again requested (line 40).

RUN the program at least five times, testing the input values of 0, 1, 2, 3, and a number greater than 3. To end the program press the RUN/STOP and RESTORE keys. To read more than one DATA statement, simply remove a flag value. For example, to read the data items in lines 80 and 90, remove the flag value (-99) from line 80, or to read the last six data items for an input value of 2, remove the flag value (-99) at the end of line 90.

To edit this program to use a FOR NEXT loop to read the data values,

```
ENTER:    50 FOR I=1 TO 3
          60 READ B:PRINT B
          70 NEXT I
```

RUN this program, using the values 0, 1, 2, 3, and a number greater than 3 as input at least once. Press the RUN/STOP and RESTORE keys to end the program. This program provides the same output as the previous program. Of course, the flag value in each of the DATA statements is no longer needed, but will not be read because the upper limit of the FOR statement is 3.

To demonstrate a valid programming use for RESET, the following program would be of use to automobile salesmen if they are wise enough to have a Commodore 64 in their showroom. This program lists the features available for a jeep. The options for this jeep, which salesmen would love to sell every buyer, are pictured in Figure 11.1.

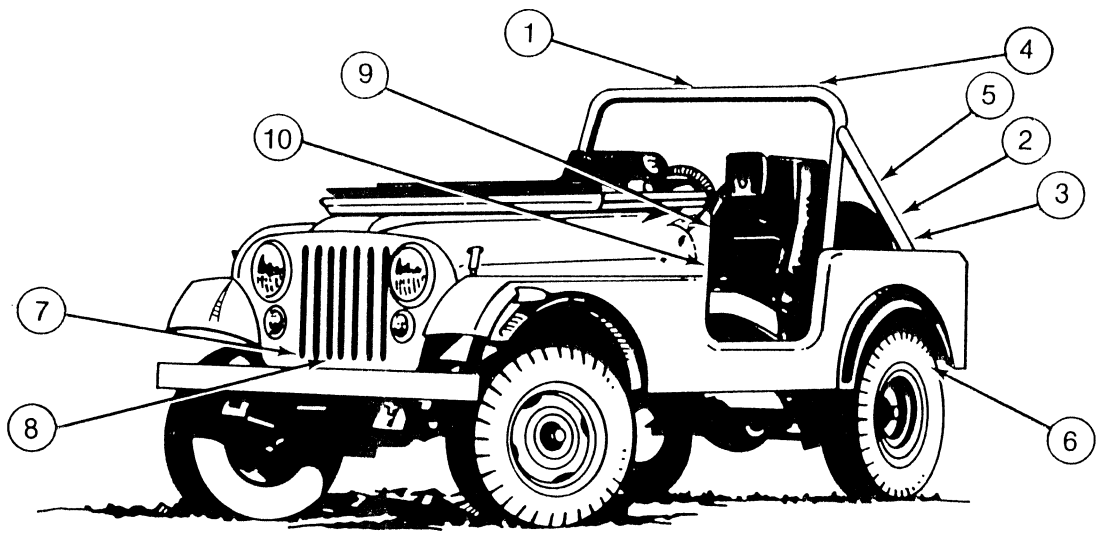


Figure 11.1: JEEP showing 10 buyer options.

Clear computer memory, use AUTO 10,10 for automatic line numbering, and enter the following program. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
CENTRE	10 PRINT "[CLEAR DOWN]":CENTRE "AUTO FACTS":PRINT "[DOWN2]"
	20 INPUT "NAME OF AUTO";CAR$

```
IF THEN ELSE   30 IF CAR$<>"JEEP" GOTO 200:ELSE:PRINT
               40 PRINT "LIST STANDARD FEATURES (STD)
               OR OPTIONAL (OPT) FEATURES?":PRINT
               50 INPUT "STD OR OPT";A$
RESET          60 IF A$="STD" THEN RESET 110
RESET          70 IF A$="OPT" THEN RESET 130
               80 READ D$,P:IF D$="END OF LIST"
               GOTO 150
               90 PRINT D$:IF P>0 THEN PRINT TAB(5) P
               100 GOTO 80
               110 DATA 4.2 LITER ENG.,,4-SPD. MANUAL
               FLR. SHIFT,,CONSOLE,,SPARE,0
               120 DATA H.D. ENGINE PKG.,,HARDTOP,,
               END LIST,0
               130 DATA (1)CONVERTIBLE TOP,379,(2)TOP
               BOOT,28,(3)TIRE COVER,26,(4)ROLL BAR
               PAD
               140 DATA 71,(5)SADDLE BAGS,52,(6)
               SPLASH GUARDS,28,(7)GRILL GUARD,67
               150 DATA (8)WINCH ASSEMBLY,718,(9)AIR
               CONDITIONING,710,(10)AM/FM RADIO,115
               160 DATA END OF LIST,,
IF THEN ELSE   170 IF A$="OPT" GOTO 200:ELSE:PRINT
               180 INPUT "SEE OPTION LIST (Y/N)";ANS$
               190 IF ANS$="Y" THEN A$="OPT":GOTO 70
               200 END
```

This program is written to show the standard and/or optional features of only one car—the JEEP. If JEEP is not entered as input for NAME OF AUTO, the program will end. Of course, a car dealer would include a menu listing all of the cars that he sells, and use INKEY for function key branching.

RUN the program, testing all of the program options, that is, auto other than JEEP, standard listing only, standard and options listings, and options listing only. **Note:** The double commas in the DATA statements on lines 110 and 120 indicate no value, that is, a zero for a numeric variable or an emply string for a string variable. This convention cannot be used at the end of a statement line when another DATA statement follows.

Using a computer as a sales tool in a dealership showroom would be exciting for automobile buyers and would eliminate possible salesmen

errors on available options and option prices. Of course, a picture of the buyer's dream car on the computer screen—in the exterior and interior colors of their choice—could be a terrific sales tool also! The software's available for this challenge, too.

LOCAL and GLOBAL

These commands permit the same variable (variable name) to be used for holding different data values in different parts of the same program. The LOCAL statement stores the values of previously defined variables (assumed to be GLOBAL) and clears the values for reuse. The local values can then be stored and cleared, and the global values returned with the use of the GLOBAL statement.

Statement format:
LOCAL variable list
GLOBAL

LOCAL must specify all variables that are to be reused. These variables must have been previously declared. If a variable is specified in the variable list of LOCAL that has not been previously used in the program, the program will "hang." To break out of the program you must press the RUN/STOP and RESTORE keys. If LOCAL is part of a multiple statement line, it must be the *last statement* on that line. If any statement follows LOCAL on a statement line, it will cause a SYNTAX ERROR.

The GLOBAL statement may be used anywhere on a multiple statement line. It will restore the values of all variables previously declared as LOCAL to their original values—global values. You may switch back and forth between the global and local values throughout a program.

Enter the brief demonstration program shown below, using AUTO 10,10 to generate line numbers, or use those shown in the listing.

```
10 A=10:A$="DIVISION 1":PRINT A,A$
20 LOCAL A,A$
30 A=30:A$="DIVISION 2":PRINT A,A$
40 GLOBAL:PRINT A,A$
50 LOCAL A,A$
60 PRINT A,A$
70 END
```

RUN the program. The output shows first the GLOBAL values (line 1Ø), then the new LOCAL values set in line 3Ø. The original values are then restored by GLOBAL in line 4Ø. Finally they are changed back to the LOCAL values again in line 5Ø. The screen output is:

```
1Ø      DIVISION 1
3Ø      DIVISION 2
1Ø      DIVISION 1
3Ø      DIVISION 2
```

The next program further demonstrates LOCAL and GLOBAL by using five variables (three numeric and two string variables) with INPUT, LET (assignment), and the READ and DATA statements assigning the initial values. Four of the five variables will be used both as LOCAL and GLOBAL variables. Use AUTO 1Ø,1Ø or key in the line numbers when entering the following program. The Simons' BASIC commands illustrated are shown to the left of the program listing at the line in which they occur.

Command Illustrated	Program Listing
	1Ø PRINT "[CLEAR DOWN]"
	2Ø INPUT "ANY NUMBER BETWEEN 1 AND 5";N
	3Ø A=1ØØ:TYPE$="GLOBAL ":V$="VALUES"
AT	4Ø PRINT:PRINT AT(25,6)TYPE$;V$
	5Ø PRINT "N=","A=","B="
	6Ø FOR I=1 TO 3:READ B:PRINT N,A,B:NEXT I
LOCAL	7Ø LOCAL N,A,B,TYPE$
RESET	8Ø TYPE$="LOCAL ":IF N=Ø THEN RESET 15Ø
AT	9Ø PRINT:PRINT AT(25,12)TYPE$;V$
	1ØØ FOR I=1 TO 3:READ B:A=A+1:PRINT
	N,A,B: NEXT I
GLOBAL	11Ø GLOBAL
AT	12Ø PRINT:PRINT AT(25,16)TYPE$;V$
	13Ø PRINT N,A,B
	14Ø DATA 8,9,1Ø
	15Ø DATA 11,12,13
LOCAL	16Ø LOCAL N,A,B,TYPE$
	17Ø PRINT:PRINT AT(25,18)TYPE$;V$
	18Ø PRINT N,A,B
	19Ø END

RUN the program. Study the program output carefully to understand exactly how the variable values were treated throughout the program. When variables are designated as LOCAL the previous value is cleared. If another value is not assigned, the value of the variable remains Ø if numeric or null if a string variable. The variable N in this program is not assigned a value as a LOCAL variable and remains Ø. This new value of zero is then used as the test to RESET the data pointer.

If you would like to run this program several times without entering RUN, replace the END statement with: **RESTORE:PAUSE 5:GOTO 2Ø**. These statements will cause the program to restore the original data pointers, pause for five seconds, and then repeat the program. The INPUT prompt will appear at the line following the program output. Because of the AT statements (lines 4Ø, 9Ø, 12Ø), the output will print in the same screen locations, replacing the numbers previously output. Each time the INPUT value is covered by the blinking cursor, you may input another value. The only data that will change will be the value of the INPUT variable, N. Press the RUN/STOP and RESTORE keys to end the program.

This ends our work with Simons' new methods of data handling.

12 Structured Programming

Here are extremely useful additions to the BASIC programming language—additions that make programming more efficient. These are statements that you will use frequently.

This chapter reviews the four Simons' BASIC procedure commands and the MERGE command. A "procedure" is a program segment that, like a subroutine, can be used more than once during the execution of a program. The MERGE command enables you to add a program or a "procedure" from a storage device to the program in memory. It is reviewed here because of its use in establishing and using a procedure "library."

The four procedure commands help to change Commodore BASIC into a more structured programming language—for a touch of Pascal! PROC, END PROC, and EXEC will replace your use of GOSUB and RETURN. PROC and CALL will sometimes replace your use of GOTO. Procedures, unlike Commodore BASIC subroutines and GOTO, perform a branch using their procedure name, not a line number. This is the power of PROCs—the ability to branch without using a line number.

The MERGE command adds a program from a storage device to the program currently in memory, providing a method of using previously written programs, subroutines, or procedures to create a new program, or a new version of a program. This, of course, can save a great deal of programming time.

There are two types of Simons' BASIC procedures: closed and open-ended. The closed procedure can replace a subroutine, Commodore BASIC's GOSUB and RETURN statements, by using the procedure statements PROC, END PROC, and EXEC. The open-ended procedure can be used instead of the BASIC GOTO statement, using the proce-

dure statements PROC and CALL. Both types of procedures eliminate reference to line numbers by using the PROC statement to reference a procedure name. Using a character string reference for branching, rather than a line number, has obvious advantages. It eliminates the coding nightmare of having used an incorrect line number—or two—for GOTO's or GOSUB's, then searching through the program for that correct line number. A descriptive name is, of course, far easier to remember than a line number.

We will begin our review of Simons' BASIC procedures with the open-ended procedure, which may be used in place of the BASIC GOTO statement.

PROC

The PROC statement assigns a user-specified name to a subroutine. This procedure name enables program transfer to be executed without regard to line numbers. The procedure name, rather than a line number, is specified by the calling statement.

Statement format:
PROC procedure name

The procedure name is assumed to be all characters following the PROC command. Therefore, on a multiple statement line PROC must be *the last statement*. The length of the procedure name is only limited by the length of a statement line (two print lines).

An open-ended procedure for averaging is shown below. Enter these statement lines now for use with the next program.

```
80 PROC AVERAGE
90 AVERAGE=SUM/COUNT
100 PRINT "AVERAGE:";AVERAGE
```

A branch to this procedure is performed by the CALL statement.

CALL

This statement is used in place of GOTO, providing the procedure name needed for transferring program control.

Statement format:
CALL procedure name

With CALL replacing GOTO and procedure name replacing line number, the statements are equivalent. After the branch is executed, the program proceeds sequentially, by line number, the same as it would following the GOTO statement. The CALL statement, like the PROC statement, must be the *last statement* on the statement line, since all characters following CALL are read as part of procedure name. Of course, the procedure name used in the CALL statement must be exactly the same as the procedure name of the PROC being branched to.

The following demonstration program will use the procedure, AVERAGE, that was entered previously, and create and use a second procedure, ACCUMULATE DATE. To enter this program use the line numbers shown. (If automatic line numbering is used, line 8Ø will be removed!) The Simons' BASIC commands illustrated are shown to the left of the program listing.

Commands Illustrated	Program Listing
CENTRE	1Ø PRINT:CENTRE "AVERAGING":PRINT
PROC	2Ø PROC ACCUMULATE DATA
	3Ø INPUT "NUMBER";N
CALL	4Ø IF N=-99 THEN CALL AVERAGE
	5Ø SUM=SUM+N:COUNT=COUNT+1
	6Ø PRINT N
	7Ø CALL ACCUMULATE DATA

LIST the program. The procedure named AVERAGE comprises the last three program lines. RUN the program, using as many input values as you wish. To obtain the average of the input values and end the program, use -99 as input.

One advantage to using PROC and CALL, rather than GOTO, is to avoid referencing line numbers. This would be a greater advantage in a program where AVERAGE was accessed from many different points in the program. A second advantage is that PROC AVERAGE can be stored as a separate program and tacked on to this or another program

using MERGE, without regard to line numbers. This will be demonstrated later in this chapter.

Now we will review closed procedures, the more powerful and useful of the two procedure methods. The three statements—PROC, EXEC, and END PROC—perform the same programming function as GOSUB and RETURN, but use a procedure name for branching rather than a line number. The PROC statement which was already reviewed for open-ended procedures, uses the same format for closed procedures.

EXEC

This statement is the branching (sending) statement for closed procedures.

Statement format:
EXEC procedure name

Again, this statement must be the *last statement* on a multiple statement line, because all characters following EXEC are assumed to be a part of the procedure name. Of course, the procedure name used in EXEC must be exactly the same as the procedure name used for the PROC statement to which program control is being transferred.

END PROC

The END PROC statement, the last statement of a closed procedure, transfers program control back to the statement following the EXEC statement that sent it.

Statement format:
END PROC

For a demonstration of closed procedures enter the following program, either using AUTO 10,10 or entering the line numbers shown. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Commands Illustrated	Program Listing
CENTRE	10 PRINT "[CLEAR DOWN]":CENTRE "SOLVE THE MATH PROBLEM":PRINT
	20 PRINT:PRINT "ENTER:":PRINT TAB(5) "1 FOR ADDITION"
	30 PRINT TAB(5) "2 FOR SUBTRACTION"
FETCH	40 FETCH "[DOWN]",1,A
IF GOTO ELSE	50 IF A<1 OR A>2 GOTO 20:ELSE:
EXEC	EXEC RANDOM NUMBERS
IF THEN ELSE	60 IF A=1 THEN Z=X+Y:PRINT X;" + ";Y;" = ":ELSE:Z=X-Y:PRINT X;" - ";Y;" = "
EXEC	70 EXEC INPUT ANSWER
CENTRE	80 PRINT "[DOWN2]":CENTRE "NEXT PROBLEM":PRINT
	90 GOTO 20
	100 END

This main portion of the program will EXECute two procedures: RANDOM NUMBERS and INPUT ANSWER. We can now simply add them to the end of this program, since we don't need to be concerned with line number references.

To enter these procedures either use AUTO 1000,10 for automatic line numbering or enter the line numbers as shown in the following listing. Again, the Simons' BASIC commands illustrated are listed to the left of the statements in which they occur.

Command Illustrated	Procedure Listing for Random Numbers
PROC	1000 PROC RANDOM NUMBERS
	1010 REM X=RANDOM NUMBER IN RANGE 40-99 and Y=RANDOM NUMBER IN RANGE 10-40
	1020 X=INT(60*RND(1))+40
	1030 Y=INT(31*RND(1))+10
END PROC	1040 END PROC

Note: For those of you who have had trouble using RND to set various number ranges, this formula may be helpful: lowest number in range minus 1, subtracted from the highest number in range equals the

multiplier. In the above program, the multiplier for X is (4Ø-1)-99, or 99-39, which equals 6Ø, and the multiplier for Y is (1Ø-1)-4Ø, or 4Ø-9, which equals 31.

Enter the second procedure, INPUT ANSWER, to complete the program. Again, the Simons' BASIC commands illustrated are listed to the left of the statements in which they occur.

Command Illustrated	Procedure Listing for Input Answer
PROC	1Ø5Ø PROC INPUT ANSWER
	1Ø6Ø INPUT "ANSWER";ANS
IF THEN ELSE	1Ø7Ø IF ANS=Z THEN PRINT "CORRECT":ELSE:PRINT "INCORRECT. THE ANSWER IS:";Z
END PROC	1Ø8Ø END PROC

RUN the program and end by pressing the RUN/STOP and RESTORE keys.

Each of these procedures can be saved on a storage device and be added to future programs, using the MERGE command. This will be shown in the next demonstration program which will add two previously saved procedures to a main program. This program, Master Grade List, will be of particular interest to both teachers and students alike. It is used to list and average grades, convert number grades to letter grades, and display letter grade distribution of student test scores. The two procedures that will be used are: GRADE AVERAGE and GRADE CONVERSION.

Enter the first procedure, GRADE AVERAGE, using AUTO 1ØØØ,1Ø or the line numbers shown.

```
1ØØØ PROC GRADE AVERAGE
1Ø1Ø ROUND=SUM/6
1Ø2Ø GRADE=INT(ROUND)
1Ø3Ø END PROC
```

If you have a storage device, disk drive or cassette drive, SAVE this procedure using "GRADE AVERAGE" as the program name, and clear computer memory.

Enter the second procedure, GRADE CONVERSION, using AUTO 2ØØØ,1Ø or the line numbers shown.

```
2000 PROC GRADE CONVERSION
2010 IF GRADE<=64 THEN F=F+1:LG$="F": GOTO 2070
2020 IF GRADE<=73 THEN D=D+1:LG$="D": GOTO 2070
2030 IF GRADE<=79 THEN C=C+1:LG$="C": GOTO 2070
2040 IF GRADE<=87 THEN B=B+1:LG$="B": GOTO 2070
2050 IF GRADE <=93 THEN B1=B1+1:LG$="B+": GOTO 2070
2060 IF GRADE<=100 THEN A=A+1:LG$="A": GOTO 2070
2070 END PROC
```

SAVE this procedure using the program name "GRADE CONVERSION" and clear computer memory.

Enter the following main program using AUTO 10,10 or the line numbers shown with the listing. The Simons' BASIC commands illustrated in this program are shown to the left of the program listing.

Command Illustrated	Program Listing
CENTRE	10 PRINT "[CLEAR DOWN]":CENTRE "MASTER GRADE LIST-ENGLISH 1-J.PARKER":PRINT
AT	20 PRINT AT(5,6)"TEST SCORE GRADES":PRINT
IF GOTO ELSE	30 READ N$:IF N$="END" GOTO 140 :ELSE:PRINT N$
	40 CT=0:SUM=0
IF GOTO ELSE	50 READ G:IF G=-99 GOTO 60:ELSE:PRINT G,:CT=CT+1:SUM=SUM+G:GOTO 50
EXEC	60 EXEC GRADE AVERAGE
EXEC	70 EXEC GRADE CONVERSION
	80 DATA JANE HALL,83,84,85,92,100,96,-99
	90 DATA SAM BENDER,93,88,89,100,70,100,-99
	100 DATA JOE PRIME,75,84,73,82,78,76,-99
	110 DATA END
	120 PRINT:PRINT "GRADE AVERAGE:";GRADE; "LETTER GRADE:";LG$
	130 PRINT:GOTO 30
PAUSE	140 PAUSE 5
	150 PRINT:PRINT "NUMBER OF STUDENTS RECEIVING EACH GRADE IS:"
	160 PRINT "A=";A,"B+=";B1,"B=";B; "C=";C; "D=";D;"F=";F
	170 END

MERGE

This command loads a program from a storage device into memory, adding the stored program *to the end of the program in memory.* It does not merge the two programs by line number. You should always list the program after using MERGE. Depending on the programs merged, you may wish to use RENUMBER or simply renumber certain statement lines. When merging a procedure onto a main program, this should not be a problem. Simply number your procedure statement lines using much higher line numbers than would normally be used in your main programs.

Command format:
MERGE "program name",device number

The MERGE parameters program name and device number follow the same syntax rules as Commodore BASIC's SAVE and LOAD commands. Program name is limited to 16 characters, and device number 8 specifies a disk drive, while no device number or device number 1 specifies a cassette unit.

If you were able to SAVE the two procedures before entering the main program above,

ENTER: `MERGE "GRADE AVERAGE",8`

if your storage unit is a disk drive.

ENTER: `MERGE "GRADE AVERAGE",1`
or
`MERGE "GRADE AVERAGE"`

if your storage device is a cassette unit.

LIST the program. The first procedure, GRADE AVERAGE, has been added to the current program. Add the second procedure in the same manner.

ENTER: `MERGE "GRADE CONVERSION",8`

if you're using a disk drive.

ENTER: `MERGE "GRADE CONVERSION",1`
or
`MERGE "GRADE CONVERSION"`

if you're using a cassette unit.

LIST the program. Both procedures are now appended to the main program. RUN the program.

Of course MERGE is not limited to adding procedures to main programs. You can merge any program to the program in memory. Just remember that the program added from your storage device will be added to the end of the program in memory—with line numbers unchanged. MERGE *does not* combine the two programs by line number. You may have to list and repair!

All programmers will certainly use PROC's and MERGE to simplify their programming tasks, whether those tasks are performed at the office, or at home for personal business or pleasure.

IV Low-Resolution Color and Graphics

Introduction

The following is a brief overview, by chapter, of the low-resolution color, graphic, and special screen commands. The low-resolution screen is the normal Commodore screen, 40 columns wide—numbered 0 through 39, by 25 rows deep—numbered 0 through 24. This is the screen mode that has been used so far in this book.

Chapter 13

COLOUR sets both the border and screen colors, replacing the Commodore BASIC commands POKE 53280 for border color, and POKE 53281 for screen color.

BCKGNDS permits the user to specify the background color of non-graphic characters. Up to three different character background colors can be displayed by using special formats in PRINT commands.

FLASH enables the user to flash the character background color by alternating between normal and reverse field at specified intervals.

OFF turns off the FLASH command.

BFLASH is used to flash the screen border by alternating between two user-specified colors displayed at specified intervals.

BFLASH 0 turns off the BFLASH command.

FILL covers a specified screen area with a designated character, displayed in a specified color.

FCOL is used to change the print color of characters.

FCHR specifies characters to be printed in a specific screen area. This command is used with FCOL to perform the same function as FILL.

SCRSV is the statement used to save a screen display on a storage device.

SCRLD is the command used to restore a previously saved screen display.

Chapter 14

MEM moves the Commodore character set from ROM into RAM so that they may be replaced with user-defined characters.

DESIGN 2 specifies the Commodore character to be replaced with a user-defined character.

The @ command is used to create ("draw") the user's graphic character.

LEFT, RIGHT, UP and DOWN are the screen "scrolling" commands. They are used to scroll a specified screen area, thereby moving the characters displayed in that area.

Key to Entering Program Listings

Additional key conventions used in this section and throughout the rest of the book are:

[RVS ON] means to hold down the control key and press the RVS ON key.

[SHIFT LOCK] means to press the SHIFT LOCK key, locking it in down position.

[SHIFT LOCK OFF] means to press the SHIFT/LOCK key to return it to the up position.

[CTRL] means to hold down the CTRL key while pressing the number key shown next.

13 Color and Characters

Now begins the demise of POKEs! The memory locations for screen and border colors can be forgotten when using Simons' BASIC. They are replaced with a one word command—COLOUR. Four additional commands allow other exciting uses of color on a low-resolution screen, while three new commands combine interesting uses of characters and color. There are even commands to save and load a screen display. Each command will first be demonstrated in the immediate mode, then in the program mode.

The commands reviewed in this chapter are especially suited to both entertainment and educational programs.

COLOUR

This is the only command used by Simons' two new graphic screens—high resolution and multicolor—that is also used on a low resolution screen. (The high resolution screen and its special commands are reviewed in Section V, and the multicolor screen in Section VI.)

This command sets *both* the border color and screen color, using only the command word COLOUR, and the Commodore BASIC color numbers. It replaces the two Commodore BASIC commands—POKE 53280, border color, and POKE 53281, screen color—that you have been using to set the screen and border color.

Command format:
COLOUR border color number,screen color number

Again, as with CENTRE, this command uses the English spelling of

the word color. You'll have to remember to spell this command word with the additional letter U.

For your convenience, the Commodore BASIC color numbers used for both screen and border color, are shown in Figure 13.1. I have also included the key press for each color, since you will need these for another command covered in this chapter.

Color Number	Color	Key Press
Ø	Black	CTRL 1
1	White	CTRL 2
2	Red	CTRL 3
3	Cyan (powder blue)	CTRL 4
4	Purple	CTRL 5
5	Green	CTRL 6
6	Blue	CTRL 7
7	Yellow	CTRL 8
8	Orange	CMDR 1
9	Brown	CMDR 2
1Ø	Light Red (hot pink)	CMDR 3
11	Gray 1 (dark grey)	CMDR 4
12	Gray 2 (medium grey)	CMDR 5
13	Light Green	CMDR 6
14	Light Blue	CMDR 7
15	Gray 3 (light grey)	CMDR 8

Figure 13.1

The following command will create a blue screen with a yellow border.

ENTER: `COLOUR 7,6`

To return the screen to the normal colors,

ENTER: `COLOUR 6,15`

This command will also be used in conjunction with the other color commands reviewed in this chapter.

BCKGNDS

This command changes the *character background color* of PRINT command characters that are entered in one of three specific formats.

Those formats are: SHIFTed characters, RVS ON characters, and RVS ON SHIFTed characters.

As you know, each character resides in a one-byte square consisting of 8 by 8 pixels. Each character fills only a portion of this square. The character background is that portion of the square not covered by the character. You clearly see this only when a character is printed in reverse field, that is, when using the RVS ON key, because the character background is a different color than the screen.

Command format:
BCKGNDS sc,b1 color,b2 color,b3 color

The parameter sc designates the screen color. This will be the background color of that portion of the screen which is not filled by characters using the BCKGNDS command.

The parameters b1 color, b2 color, and b3 color designate the color to be used for the background of characters entered in one of the three special PRINT formats. The color numbers used for each of the 16 Commodore colors are shown in Figure 13.1 on page 108.

The special PRINT formats for the BCKGNDS parameters are shown below.

b1 color: The background color for all SHIFTed characters.
b2 color: The background color for all RVS ON characters.
b3 color: The background color for all RVS ON SHIFTed characters.

All four parameters of the BCKGNDS command must be specified, even if all three color parameters will not be used. The color parameter for formats that are not used will simply be ignored.

Only non-graphic characters—those shown on the key tops—can be used by this command. The graphic characters shown on the key fronts cannot be used. Single character keys, such as alphabetic characters, can use any of the three text formats—shifted, reverse field, or reverse field and shifted. Keys containing two characters, for example, the number keys on the top row and the five punctuation keys on the third and bottom rows, can use **only** the reverse field format. I know that this doesn't seem reasonable, since the upper character must be shifted to print, but, trust me, it's true!

All of the above will be infinitely clearer after you have used this command. Remember, the first parameter of the BCKGNDS command

is screen color, not a character background color. The order of the background color formats is: *shifted, reverse field, reverse field shifted.*

Each of the following immediate mode examples will print the character string "COMMODORE" on a yellow screen.

The first example is printed in the RVS ON format. Therefore, the characters will use the b2 color, color number 4 (purple).

```
ENTER:   BCKGNDS 7,3,4,5:PRINT "[RVS ON]COMMODORE"
```

In the next example the character background will use the b1 color, color number 3 (cyan) because the format is SHIFTed characters.

```
ENTER:   BCKGNDS 7,3,4,5:PRINT[SHIFT/LOCK]
         "COMMODORE" [SHIFT/LOCK OFF]
```

The third example, shown below, uses the RVS ON SHIFTed format accessing the third color parameter, b3, the color green, color number 5.

```
ENTER:   BCKGNDS 7,3,4,5:PRINT "[RVS ON]
         [SHIFT/LOCK] COMMODORE"[SHIFT/LOCK OFF]
```

To restore the normal screen colors, press the RUN/STOP and RESTORE keys.

I deliberately used reverse field in the first example above in order to present this format and the command without confusion. If shifted characters had been the first entry, the characters that printed on the screen might well have confused you! This is explained below.

Important Note: When entering shifted characters, they print the graphic characters shown on the key fronts. The Commodore is using the upper case/graphics mode when either the SHIFT or SHIFT/LOCK key is used. (In fact, the typewriter mode—upper/lower case—is not accessible when using Simons' BASIC.) For this command, entering text in the SHIFTed format **when a background color is initialized** (used for the first time) will cause the graphic character set to be printed. The graphic characters will be changed to letters as soon as the commands are entered, producing the correct results. After a color has been initialized, characters typed in the SHIFTed format will print the characters shown on the key top. The characters you expected to see printed! Of course, until the color is initialized, there is a serious disadvantage in checking typing accuracy! This disadvantage is more serious in the program mode than in the immediate mode. However, now you won't be surprised when you see graphic characters instead of the characters you thought you were entering. This scenario will be shown in the brief demonstration program that follows.

This program uses the three background formats, with each using a different color. When the text COMMODORE and COMPUTERS is entered, they will print as graphic characters. The Simons' BASIC commands demonstrated are listed to the left of the program statements.

Command Illustrated	Program Statements
	10 PRINT "[CLEAR]"
COLOUR	20 COLOUR 6,12
BCKGNDS	30 BCKGNDS 12,3,4,7:PRINT [SHIFT LOCK] "COMMODORE[SHIFT LOCK OFF][RVS ON] 64 [SHIFT LOCK]COMPUTERS!"[SHIFT LOCK OFF]

Since RVS ON was not turned off in line 30, COMPUTERS will use the color for the reverse field shifted format.

RUN the program. You certainly have vivid color! You may want to change the color numbers to tone down the contrasts. Do you see the background color of the exclamation mark? Even though the SHIFT key was also used, because this character resides on a two-character key it can only use the reverse field format. The only keys that can access either of the two shifted formats (SHIFTed and RVS ON SHIFTed) are the character/graphic keys. This is important to remember when planning your character backgrounds.

Press the RUN/STOP and RESTORE keys to return to the normal screen.

Now you will see why a typing error can be a real problem when initializing a shifted background color. LIST the program. The graphic characters have **not** been replaced by alphabetic characters in the listing. I don't think you would want to edit this PRINT statement the way it now appears! But, there's a remedy.

RUN the program again and **do not** clear the screen. LIST the program. Now the program can easily be edited. Since seeing is believing, use the editing keys to replace the word COMPUTERS in line 30 by pressing SHIFT/LOCK and entering WITH SIMONS BASIC. RUN the program once again. **Note:** If you used an apostrophe at the end of SIMONS, it printed in purple—used the b2, reverse field format color. Remember, characters on two-character keys can only use the reverse field format. Now that you understand these possible problems, we will move on to another.

If more than one BCKGND statement is used in a program, the color parameters of the most recently executed BCKGND statement will over-

ride the color parameters of the preceding BCKGND statement(s). The following statements added to the program in memory will demonstrate this. Clear your screen (press the RUN/STOP and RESTORE keys), LIST the program, and enter the following statement lines.

```
40 PAUSE 3
50 PRINT "[DOWN 2]"
60 BCKGNDS 12,4,5,6:PRINT [SHIFT LOCK]"CAN [SHIFT
LOCK OFF][RVS ON]BE [SHIFT LOCK]COLORFUL" [SHIFT
LOCK OFF]
```

RUN the program. The background of the first character string was changed to the colors specified in the second BCKGNDS statement—purple, green, and blue—as soon as the second BCKGNDS statement (line 60) was executed. If you wish to use different background color parameters in the same program, clear the screen between BCKGND statements.

Press the RUN/STOP and RESTORE keys to return to a normal screen, and clear computer memory.

The next command gets even flashier!

FLASH

This command automatically uses the Commodore RVS ON and RVS OFF functions, enabling you to alternate between normal and reverse background color for any PRINT command character, graphic or non-graphic, that has been entered in the Commodore color specified in this command. You may use both the CTRL and CMDR color keys, making use of all 16 colors. For the Commodore color numbers and their associated key press, refer to Figure 13.1, page 108.

Command format:
FLASH color number
or
FLASH color number,speed

When a speed is not specified, "flashing" will occur every four seconds. To set a specific speed, specify a number in the range 1 through 255. Each unit of speed is approximately one-sixteenth of a second. A speed of 80 would cause flashing to occur every five seconds, which is

quite slow. A rapid speed will occur by using ten or less for the speed parameter.

The same color must be specified in the FLASH command as the color used in the PRINT command of the character(s) to be flashed. For example, using CMDR 7 in the PRINT command would require using the color number 14 in the FLASH command. All characters on the screen having the color specified in FLASH will alternate between normal and reverse-field in that color. Be sure to use a different color, or no color, on those characters that you don't want "flashing". After the FLASH command is initialized, "flashing" characters will continue to be displayed unless FLASH is turned off. Even when the program ends, the word READY will flash, and all characters entered thereafter will flash. So it is a must to learn how to turn flashing off.

Command format:
OFF

Fortunately, this is an easy command to remember. Both FLASH and OFF can be used in the immediate mode.

The following set of commands will demonstrate the FLASH and OFF commands.

```
ENTER:    PRINT "[CTRL 3]SIMONS' [CTRL 8]BASIC":
          FLASH 2,10:PAUSE 3:OFF
```

Only the character string entered in red flashed, because this is the only color specified in a FLASH command. To also flash the word BASIC, you must enter a second FLASH command, **FLASH 7,10**. The new set of commands is shown below. Clear the screen before entering these commands.

```
ENTER:    PRINT "[CTRL 3]SIMONS' [CTRL8]BASIC":
          FLASH 2,10:FLASH 7,10:PAUSE 3:OFF
```

Now the entire string flashes—SIMONS' in red and BASIC in yellow.

Note: Be careful not to use either the screen or border color in a FLASH command. Using the screen color for characters will, of course, produce nothing. Using the border color will cause the entire screen to "flash"!

When using both the BCKGNDS command and the FLASH command in one program, some advance planning must be done. Both the second

background color parameter of the BCKGNDS command and the FLASH command use RVS ON. The RVS ON color parameter in BCKGNDS can override the FLASH command color.

If the BCKGNDS statement *precedes* the color statements to be used by FLASH, the background will flash in the color specified by the RVS ON color parameter in the BCKGNDS statement, ignoring the color specified in FLASH. The characters will be printed in the color specified in the print statement—but "flash" the background color specified in the RVS ON parameter of the BCKGNDS statement. If the RVS ON color parameter of the BCKGNDS statement and the FLASH command color are the same, the characters will alternately be covered by a solid bar in the specified color and printed in the specified color. This is not really the same as "flashing" characters, as the characters are not always visible. This is fine as long as this is the effect that is wanted!

If the BCKGNDS statement follows the color statements to be used by FLASH, the background color used in those statements is not affected. However, remember that the print color used in the last print statement will continue to be used throughout the program, unless it is specifically changed either in a separate statement or as part of a print statement.

The FLASH statement may appear anywhere in the program. Placement of this statement does not affect the statement's execution.

To demonstrate a flashing character background color in a program, enter the following listing. For automatic line numbering, first use AUTO 10,10. The Simons' BASIC commands illustrated are shown to the left of the program statements.

Command Illustrated	Program Statements
	10 PRINT "[CLEAR]"
COLOUR	20 COLOUR 6,12
	30 PRINT "[DOWN 3]"
BCKGNDS	40 BCKGNDS 12,3,4,7:PRINT [SHIFT LOCK] "COMMODORE[SHIFT LOCK OFF][RVS ON] 64 [SHIFT LOCK]COMPUTERS"[SHIFT LOCK OFF]
AT	50 PRINT AT(12,12)"[CTRL 5]ARE MORE EXCITING"
AT	60 PRINT AT(18,14)"[CTRL 7]USING"
AT	70 PRINT AT(15,16)"[CTRL 7]SIMONS' BASIC"

```
FLASH       80 FLASH 4,10
PAUSE       90 PAUSE 5
OFF         100 OFF
```

RUN the program and press the RUN/STOP and RESTORE keys to return to the normal screen.

This program used the same color—purple (4)—in both the RVS ON parameter of the BCKGNDS statement and the color number parameter of the FLASH statement. This produced a solid bar of purple alternating with the display of the character string "ARE MORE EXCITING".

LIST the program. To flash a different background color for the character string "ARE MORE EXCITING", change the RVS ON color option in the BCKGNDS statement. For example, using the color number 1 instead of 4 will cause the background color to flash white rather than purple. The background color of the number 64 will also be changed to white. Make this color change (from 4 to 1) in line 40 and RUN the program again. Press the RUN/STOP and RESTORE keys to return to a normal screen. Using a different background color created actual "flashing," where the characters are always visible. This can also be a very effective visual technique.

And, there's more "flashing" to come!

BFLASH

This command displays two alternating border colors at a user-specified speed.

Command format:
BFLASH speed,c1,c2

The flashing speed is the same for this command as for the FLASH command, but in this command it is a required parameter. Using a range of 1 through 255, each unit is approximately one-sixteenth of a second. Color 1 (c1) and color 2 (c2) may be any one of the sixteen Commodore color numbers. When flashing ends, the border will continue to be displayed in color 2. To change the BFLASH border color, the COLOUR command must be used. The border color will continue to flash until the BFLASH command is cancelled, as shown below.

Command format:
BFLASH Ø

Using the parameter Ø cancels the border flashing command.
For a demonstration of a flashing red and blue border,

ENTER: `BFLASH 4Ø,2,6`

Then

ENTER: `BFLASH Ø`

to turn off flashing. The border will now appear in the last color flashed.

To add a flashing border to the program in memory, enter the following program statement.

```
75 BFLASH 4Ø,6,4:PAUSE 5:BFLASH Ø
```

This statement line will cause the border to flash blue then purple at intervals of approximately two and one-half seconds, for a period of five seconds. After the border flashes, the characters will flash.

RUN the program, and press the RUN/STOP and RESTORE keys to return to a normal screen.

FILL

This command fills a specified screen area with a character, and displays that character in a user-specified color. All parameters are controlled by the programmer. FILL requires two screen locations, the starting row and column coordinates. Remember, the low-resolution screen is configured Ø through 39 columns and Ø through 24 rows.

Command format:
FILL row,column,width,depth,character code,color number

Both the row and column parameters specify the **starting position** of the character to be displayed. Width specifies the total number of columns to be covered, and depth the total number of rows to be covered. The character code is the Commodore poke code for the character to be

displayed. These poke codes are listed in Appendix E of your *Commodore User's Guide*, pages 132-134. The color numbers are listed on page 108, in Figure 13.1 of this book. All six parameters are required. If any are omitted, the error message BAD MODE will be given. This error message is also generated if width plus starting column is greater than 40, or the starting row plus depth is greater than 25.

Clear your screen, and move the cursor down two screen lines, for several demonstrations of the FILL command. The first command will create a yellow border across the top of your screen, using the CMDR graphic character with the poke code 92. The character will begin in the first screen column—0—and print in a total of 40 columns. The next commands entered will present hanging strips of drying pasta! The brown rod will be printed in row 12, beginning in column 12, for a total of 15 columns. The pasta strips, in yellow, will also begin on row 12, to cover a portion of the rod. They will begin printing in column 17 and end in column 21 (a total of 5 columns). The strips will cover 9 rows. The last command will create an orange border across the bottom of your screen, using the SHIFTed graphic character with the poke code 86.

ENTER:
```
FILL 0,0,40,1,92,7
FILL 12,12,15,1,67,9:FILL 12,17,5,9,92,7
FILL 24,0,40,1,86,8
```

The following program will display a flag similar to our American flag. Using the asterisk—code 42—twelve columns by eight rows of white stars will be displayed on a blue background, enclosed in a red screen border. Clear your screen and computer memory, and enter the following program. To use automatic line numbering with this program, first enter AUTO 10,10. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Statements
COLOUR	10 COLOUR 2,6
	20 PRINT "[CLEAR]"
FILL	30 FILL 0,0,12,8,42,1
PAUSE	40 PAUSE 5

RUN the program and press the RUN/STOP and RESTORE keys to return to the normal screen. Clear computer memory.

Using the FILL command is certainly easier than using a series of PRINT AT and DUP commands!

FCOL

This command can be used to change the print color of a character. The background of the character square remains unchanged, whether it is the screen color or a user-specified background color.

Command format:
FCOL row,column,width,depth,color

Row and column designate the **starting location** of the character(s), width specifies the number of columns covered by the character(s), and row specifies the number of rows that the characters fill. The color specified will be the new color of the character(s) designated in this command.

Of course, you must first know the location of the characters in order to use this command. The PLACE command could be used to give the column position, but not the row.

Following are several demonstrations of FCOL, used in the immediate mode. The first set of commands prints the string "HI" beginning in column 2, row 2, pauses program execution for three seconds, then changes the color of the characters in the string from black to yellow. Clear the screen.

ENTER:
```
PRINT AT(2,4)"HI":PAUSE 3:FCOL 4,2,2,1,7
```

The next set of commands prints and flashes a character string in red, then changes the color of the characters to green.

ENTER:
```
PRINT AT(2,12)"[CTRL 3]MORE COLOR":FLASH
2,10: PAUSE 3:OFF:FCOL 12,2,10,1,5
```

The following commands will print the shifted characters in black on a red background, pause for three seconds, then change the character color to white. The background color will remain red. Clear the screen, using the RUN/STOP and RESTORE key. Remember, the character string (shifted) will print in graphic characters!

ENTER:
```
BCKGNDS 15,2,15,15:PRINT AT(2,15)
"[SHIFT LOCK] BACKGROUND COLORS"
[SHIFT LOCK OFF]:PAUSE 3:FCOL 15,2,17,1,1
```

The following program will use the FILL statement to print six, 20-

column bars in white, on a black screen created by the COLOUR statement. The first FCOL statement (line 5Ø) will change the two middle bars from white to red, and the second FCOL statement (line 7Ø) will change the second and fifth color bars from white to blue. To use automatic line numbering with the following program enter AUTO 1Ø,1Ø, then enter the program. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
	1Ø PRINT "[CLEAR]"
COLOUR	2Ø COLOUR Ø,Ø
FILL	3Ø FILL 1Ø,1Ø,2Ø,6,98,1
PAUSE	4Ø PAUSE 3
FCOL	5Ø FCOL 12,1Ø,2Ø,2,2
PAUSE	6Ø PAUSE 3
FCOL	7Ø FCOL 11,1Ø,2Ø,1,6:FCOL 14,1Ø,2Ø,1,6
PAUSE	8Ø PAUSE 3
	9Ø END

RUN the program and press the RUN/STOP and RESTORE keys to return to the normal screen.

FCHR

This command is used with the FCOL command. Together they accomplish exactly the same task as the FILL command.

Command format:
FCHR row,column,width,depth,character code

As you can see, this command does not accept a color code. That is why it needs the FCOL command in order to function. Remember that this command cannot be used alone to print characters in color, in either the immediate or program modes. If FCOL could not be used with the FILL command, then this pair of commands would serve a more valuable purpose. But since we can change all, or portions, of the characters specified in the FILL command by using FCOL, the FCHR command would seem to be a duplicate statement!

Saving and Restoring a Screen

The SCRSV (screen save) statement saves a low-resolution screen display on a storage device. The SCRLD (screen load) command is used to restore a previously saved SCRSV and redisplay the screen on your monitor. These Simons' BASIC commands **do not replace** the Commodore SAVE and LOAD commands. They are only used to save and load a specific screen display. The program instructions are not saved.

SCRSV

The screen save statements for saving to disk and to tape are shown below.

Statement format for Disk:
SCRSV 2,8,2,"screen name,S,W"

Statement format for Tape:
SCRSV 1,1,1,"screen name"

This program statement is used directly after the statements that produce the screen display to be saved. When the program is run, the program will execute normally until the SCRSV statement is executed. Then the appropriate SAVE instructions will appear on the screen. This SAVE process is the same as a BASIC program SAVE. While the screen is being saved, the screen will flash periodically. Be patient, because a screen save takes about four and a half times longer than saving a program and uses four and a half times more storage space. After completing the SCRSV statement, the program will continue normally.

If you wish to test screen save using the color bars program in memory, add the appropiate SCRSV statement to the end of the program and either **change** the screen color parameter in the COLOUR statement **or remove** this statement (line 20) from the program. You would not be able to see the save instructions on a black screen!

Remember, only low-resolution screen displays may be saved. High-resoluion and multicolor graphics cannot be saved. For this reason,

saving a screen display will probably not be used often. The most logical reason for saving a screen that I can think of would be to save the results of either an educational or game program. This could be very useful, especially for youngsters.

SCRLD

To redisplay a saved screen, use the SCRLD command.

Command format for Disk:
SCRLD 2,8,2,"name"

Command format for Tape:
SCRLD 1,1,Ø,"name"

The screen load command, unlike the BASIC LOAD command, **does not erase the program in memory** when the SCRLD command is executed.

If you saved the color bars screen, enter the appropriate SCRLD command to redisplay this screen and follow the normal Commodore LOAD instructions as they appear on the screen. The screen will flash, showing glimpses of the saved screen. When SCRLD is completed the saved screen display will appear on your monitor. Unfortunately, the SAVE instructions will appear at the top of the screen display, and READY and the cursor will appear at the bottom of the screen unless the program ended with a continuous loop. Even with this unwanted text, saving and loading a screen is certainly important for those of you who wish to save—and show off—your latest high score on a super game program that you wrote!

14 User-Defined Graphics

Three commands—MEM, DESIGN 2, and @—give you the ability to replace any character in the Commodore character set with a user-defined graphic character. These commands can only be used on a low resolution screen. Any number of Commodore characters may be redefined, creating new characters—each filling an 8 by 8 pixel square. These "new characters" may then be used in combination to create and print any size graphic, using any of the 16 Commodore colors. Three demonstration programs will be used in reviewing these commands.

After creating user-defined characters, "scrolling" will enable you to move these characters around the screen. This is accomplished with the four commands RIGHT, LEFT, UP, and DOWN. The scrolling commands will be demonstrated first through several "teaching programs", and then by adding scrolling commands to the last user-defined graphic program.

MEM

MEM automatically moves the Commodore character set from ROM (Read Only Memory) into RAM (Random Access Memory).

Statement format:
MEM

The entire character set can now be accessed. Each character can be redefined, character by character, using DESIGN 2 and @.

DESIGN 2

DESIGN 2 specifies the Commodore character to be replaced.

Statement format:
DESIGN 2,$EØØØ + character poke code * 8

DESIGN 2 is the command word. You **must** follow DESIGN with the number 2. The command DESIGN, without a number following, is used for the design of character graphics on a high-resolution or multicolor screen (Chapter 19). $EØØØ is the start address, in hexadecimal, of the Commodore character set after being moved into RAM by the MEM command. The only user-specified parameter is the poke code of the character to be replaced. This number is then added to this start address, giving the computer the address of the character to be replaced. For this reason, the + must be used in the command. Character poke codes are listed in Appendix E, pages 132-134, of your *Commodore User's Guide*. The last parameter—*8—specifies that the character is 8 by 8 pixels.

The next step is to create your own graphic character, which will replace the character specified in the DESIGN 2 character poke code parameter.

@

The at sign—@—is the command used to create each of the eight lines of the 8 by 8 grid, each representing one pixel (or dot), used to "draw" the graphic character. This is done by turning each pixel "on" or "off". To "turn on" a pixel—fill a dot—the character B is used. To leave a pixel "off" for an unfilled dot, a period is used. When a pixel is "off", nothing is printed, and only the background will appear in that dot on the screen. When a pixel is "on" that dot will be filled, causing that portion of the graphic to be filled. These 64 dots, 8 dots wide by 8 dots deep, will print one character square (one byte)—the same size as the character square that is being replaced.

The first demonstration program will replace the Commodore character Z (poke code 26) with a graphic character, designed as a stick figure.

Clear computer memory, and enter the following program. Use AUTO 100,10 for automatic line numbering. The Simons' BASIC commands illustrated are shown to the left of the program listing. Type the periods and the letter B without any spaces between them in the @ statements.

Command Illustrated	Program Listing
	100 PRINT "[CLEAR]"
COLOUR	110 COLOUR 0,1
MEM	120 MEM
DESIGN 2	130 DESIGN 2,$E000+26*8
@	140 @ . . . BB . . .
@	150 @ . . . BB . . .
@	160 @BBBBBBBB
@	170 @. . BBBB . .
@	180 @. . BBBB . .
@	190 @. . B . . B . .
@	200 @. . B . . B . .
@	210 @. . B . . B . .
AT	220 PRINT AT(5,5)"Z Z Z"
	230 END

Line 120 will move the Commodore character set from ROM to RAM, so that the characters can be accessed. Line 130 designates letter Z—poke code 26—as the character to be replaced. Lines 140 through 210 "draw" the new character—a stick figure.

RUN the program. Three stick figures will be printed on row 5, beginning at column 5. Each time the key Z is pressed the stick figure will be printed. Of course, you no longer have the use of the character Z until you press the RUN/STOP and RESTORE keys. Do this now to return to the Commodore character set. Now the character Z will again print.

Often more than one 8 by 8 pixel is needed to create a graphic character. To do this, you must draw your graphic on two or more design grids. Then by printing the new graphic characters in the correct picture order the larger graphic is correctly displayed. The following program draws a pine tree in the first design, and the tree's trunk in a second design. They are then printed, in color, positioned to display one graphic picture. Clear computer memory, and enter the following program. For automatic line numbering, enter AUTO 100,10. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
	100 PRINT "[CLEAR]"
COLOUR	110 COLOUR 6,13
MEM	120 MEM
DESIGN 2	130 DESIGN 2,$E000+26*8
@	140 @ . . . BB . . .
@	150 @ . . . BB . . .
@	160 @. . BBBB . .
@	170 @. . BBBB . .
@	180 @. BBBBBB .
@	190 @. BBBBBB .
@	200 @BBBBBBBB
@	210 @BBBBBBBB
DESIGN 2	220 DESIGN 2,$E000+25*8
@	230 @ . . . BB . . .
@	240 @ . . . BB . . .
@	250 @ . . . BB . . .
@	260 @ . . . . .
@	270 @ . . . . .
@	280 @ . . . . .
@	290 @ . . . . .
@	300 @ . . . . .
AT and DUP	310 PRINT AT(10,10) DUP("[CTRL 6]Z ",5)
AT and DUP	320 PRINT AT(10,11) DUP("[CMDR 2]Y ",5)
PAUSE	330 PAUSE 3
	340 END

Notice that the design for the tree trunk only requires a depth of three pixels. However, all eight by eight pixels must be specified. If they are not, the program will halt and the error message BAD CHAR FOR A MOB followed by the line number of the DESIGN 2 statement lacking data. MOB stands for movable object block, Simons' BASIC's name for a sprite. Sprites are also user-defined graphics, but can only be used on high-resolution and multicolor screens. Sprite commands are reviewed in Section VII.

This program will display five green trees on row 10, beginning in column 10, and the five brown trunks will print on row 11 directly below the trees. The DUP command (reviewed in Chapter 7) is useful for a multiple display of the graphic. RUN the program. Note that the Y

of READY is now a tree trunk! Press the RUN/STOP and RESTORE keys to return to the normal screen and character set.

The FILL command is also useful for displaying a border using your own graphic. The following two statement lines will add a top border of the tree graphic to the program in memory.

ENTER:

```
340 FILL 0,0,40,1,26,5
350 FILL 1,0,40,1,25,9
360 PAUSE 3
370 END
```

RUN the program again to see the added border display. Press the RUN/STOP and RESTORE keys to return to the normal screen and character set.

The last demonstration program will create a three-byte graphic, an automobile in three parts—back end, middle section, and front end. This will be done by redefining the characters X, Y, and Z. Clear computer memory, enter AUTO 100,10 for automatic line numbering, and enter the following program. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
	100 PRINT "[CLEAR]"
COLOUR	110 COLOUR 0,1
MEM	120 MEM
DESIGN 2	130 DESIGN 2,$E000+24*8
@	140 @ . . . . . BB
@	150 @ . . . . . B . .
@	160 @. BBBBB . .
@	170 @BBBBBBBB
@	180 @BBBBBBBB
@	190 @. BBBBBBB
@	200 @. . BBB . . .
@	210 @ . . . B . . . .
DESIGN 2	220 DESIGN 2,$E000+25*8
@	230 @BBBBB . . .
@	240 @ . . . BB . . .
@	250 @ . . . . BBBB
@	260 @BBBBBBBB
@	270 @BBBBBBBB
@	280 @BBBBBBBB

```
@           290 @ . . . . . .
@           300 @ . . . . . .
DESIGN 2    310 DESIGN 2,$E000+26*8
@           320 @ . . . . . .
@           330 @ . . . . . .
@           340 @BBBBBB . .
@           350 @BBBBBBBB
@           360 @BBBBBBBB
@           370 @BBBBBB . .
@           380 @. BBB . . . .
@           390 @. . B . . . .
AT          400 PRINT AT(18,12)"[CTRL 3]XYZ"
            410 END
```

RUN the program to print a sleek, red automobile in the center of your screen. Press the RUN/STOP and RESTORE keys to return to the normal screen and character set. SAVE this program! We will be adding "scrolling" to it later in the chapter.

Most graphic characters can easily be drawn on the screen grid, thanks to the @ command. Should your graphic pictures get very large, however, you will probably have to sketch them first on graph paper. Just remember to always use an 8 by 8 grid.

May your imagination and graphic abilities run wild! And the word "run" brings us to our next set of commands. But before continuing, SAVE the program in memory. It will be used again with motion added.

Scrolling Techniques

Four commands—RIGHT, LEFT, UP and DOWN—enable characters to be moved around the screen. This is not like the MOVE command in which characters are duplicated on the screen. Using scrolling, the characters move with the screen, assuming a new position.

Command format:
directionW starting row,starting column,ending column,ending row
or
directionB starting row,starting column,ending column,ending row

Direction is, of course, RIGHT, LEFT, UP or DOWN. The parameter

following the direction must be either W or B, with no space between the direction and this parameter. W specifies scrolling with "wrap-around," meaning that characters that pass off the screen will reappear on the opposite side. B specifies "blanking," meaning that characters that pass off the screen will not reappear.

The next four parameters can be confusing to use when changing direction! The key to understanding how to use these parameters is to remember that these parameters **designate the screen area to be scrolled**. They do not designate the location of the character(s) that will be moved. Everything that appears on the portion of the screen that is specified in this command will be scrolled. The starting row and starting column are the row and column coordinates of the area that is to be scrolled. The ending row and ending column are the row and column coordinates designating the end of the area to be scrolled.

For example: RIGHTB Ø,Ø,2Ø,25 specifies that scrolling to the right will begin at row Ø, column Ø and end at column 2Ø, row 25. The B following RIGHT specifies "blanking" be used, meaning that characters disappear, character by character, at the ending coordinates location. The ending row and column parameters must specify one row and column beyond where you want the scrolling to end. In the above example, scrolling will end at column 19 and row 24. Remember that columns are numbered Ø through 39 and rows Ø through 24.

The following program will demonstrate scrolling the screen RIGHT for the screen area 0,0 through 19,19 using wraparound. Clear computer memory and enter the following program. If you wish to use automatic line numbering, enter AUTO 1Ø,1Ø before entering the program lines. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
	1Ø PRINT "[CLEAR]"
AT	2Ø PRINT AT(Ø,Ø)"SCROLLING"
AT	3Ø PRINT AT(Ø,6)"THE"
AT	4Ø PRINT AT(Ø,12)"SCREEN"
AT	5Ø PRINT AT(Ø,18)"RIGHT"
PAUSE	6Ø PAUSE 2
RIGHTW	7Ø RIGHTW Ø,Ø,2Ø,2Ø
	8Ø GOTO 7Ø

During the program run, when the end of each character string meets the end of the scroll area, the last character will disappear. Because

wrap-around is being used, the character will reappear, entering at the left side of the screen. If you would like to follow the scrolling more closely,

ENTER: `75 FOR I=1 TO 100:NEXT I`

before running the program. You can vary the upper limit of I to change the speed of the scroll, increasing the upper limit for a faster scroll or decreasing the upper limit for a slower scroll. **Note:** This is not the time to use the PAUSE command, since even PAUSE 1 slows scrolling to a one-byte move per second. If you would like to see this "super slomo" replace line 75's FOR NEXT statements with PAUSE 1.

RUN the program, and press the RUN/STOP key to end the run.

LIST the program. To end scrolling at the end of the designated scrolling area, change the parameter following direction from W to **B** to use "blanking." Statement line 70 should now read as follows.

```
RIGHTB 0,0,20,20
```

RUN the program again. The screen scrolled right until all the characters disappeared from the screen. Press the RUN/STOP key to end the program.

To scroll left, edit line 70 replacing RIGHTB with **LEFTW**. No other parameters in the scrolling command need to be changed. You are not changing the screen area to be scrolled. You are simply changing direction! You will also be using wraparound this time, instead of blanking. Now change the character string in line 50 to read "**LEFT**", and RUN the program. By scrolling the screen to the left, the characters immediately begin exiting the left side of the screen. The characters re-enter at column 19, one column less then the ending column specified in the scrolling command. If the W is changed to B, blanking will occur and the characters will immediately disappear permanently off the left side of the screen. If you haven't done so already, press the RUN/STOP key to end the program. If you entered line 75 and are impatient with the slow scrolling, remove it before continuing.

To change this program to scroll DOWN, line 70 will be edited again. Two parameters must be changed—direction and ending row. If the ending row parameter is not changed, the characters will leave the screen at row 19. Changing this parameter to 25 will scroll all screen rows—0 through 24. (Remember—this parameter must be the ending row desired, plus 1.) Line 70 will now read as follows.

```
DOWNW 0,0,20,25
```

Of course you can also change the character string in line 5Ø to "DOWN".

RUN the program. Since wraparound was used, the characters will reappear at the top of the screen after scrolling off the bottom. Press the RUN/STOP key to end the program.

The last change will be to scroll the screen area UP. Since we are scrolling the same screen area, the only change needed is to change the direction from DOWN to **UP**. All other parameters remain the same. Edit line 7Ø (and line 5Ø, if you wish) and RUN the program again. Press the RUN/STOP key to end the program.

The scrolling commands may also be combined to scroll different areas of the screen in different directions. The following program demonstrates this by scrolling the top half of the screen RIGHT and the bottom half of the screen LEFT, with the screen characters printed on a diagonal. Clear computer memory and enter the following program. You may use AUTO 1Ø,1Ø for automatic line numbering with this program. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
	1Ø PRINT "[CLEAR]"
AT	2Ø FOR I=1 TO 2Ø:PRINT AT(I+1,I)"$":NEXT I
PAUSE	3Ø PAUSE 3
RIGHTW and LEFTW	4Ø RIGHTW Ø,Ø,4Ø,11:LEFTW 11,Ø,4Ø,11
	5Ø GOTO 4Ø

If you would like to slow the scrolling, add the following statement before running the program.

```
45 FOR I=1 TO 65:NEXT I
```

RUN the program and press the RUN/STOP key to end.

To have the top half of the screen scroll UP and the bottom half scroll DOWN, change RIGHTW to **DOWNB** and LEFTW to **UPB** in line 4Ø. This will also change the wraparound to blanking. RUN the program again, and press the RUN/STOP key to end.

Now you should be an expert on scrolling the screen. Scrolling can be used in many programming situations. One of the most frequent uses, however, will probably be in game programs using user-defined characters. So our last program will use scrolling to drive an automobile.

LOAD the program containing the DESIGN 2 red auto, page 126, which you saved before beginning Scrolling Techniques. LIST the program and enter the following program lines. Statement line 400 is changed to begin near the left side of the screen, rather than in the middle. You may use AUTO 400,10 for automatic line numbering. The Simons' BASIC commands illustrated are shown to the left of the program statements in which they occur.

Command Illustrated	Program Statements
AT	400 PRINT AT(10,10)"[CTRL 3]XYZ"
PAUSE	410 PAUSE 1
AT	420 PRINT AT(2,3)"[CTRL 3]GOING FORWARD!"
RIGHTB	430 FOR I=1 TO 20:RIGHTB 0,0,36,20:NEXT I
PAUSE	440 PAUSE 2
AT and PAUSE	450 PRINT AT(22,3)"[CTRL 3]IN REVERSE![SPACE 3]":PAUSE 1
LEFTB	460 FOR I=1 TO 20:LEFTB 0,0,40,20:NEXT I
PAUSE	470 PAUSE 3
	480 END

By including the scrolling statements in FOR NEXT loops, you are able to control the length of the scroll and retain the characters on the screen.

RUN the program and press the RUN/STOP and RESTORE keys to return to the normal screen.

You can also scroll only the row on which the car appears by changing both the first and last parameters—starting row and ending row—to 10 in each of the scrolling statements. This is important when you don't want the entire screen background to scroll, but only specific rows on which certain characters appear.

I'm sure you noticed the distortion of the leading character during the scrolling. This distortion can be avoided by using a delay loop to control scrolling speed.

For a demonstration of both of the above techniques, replace lines 430 and 460 as shown below.

```
430 FOR I=1 TO 20:RIGHTB 10,0,36,10:FOR J=1 TO 100:
NEXT J:NEXT I
```

```
460 FOR I=1 TO 20:LEFTB 10,0,40,10:FOR J=1 TO 100:
NEXT J:NEXT I
```

By changing the starting and ending row parameters of both scroll statements to the row on which the car is printed, only that row will scroll. The text will remain where it was originally printed. Adding the delay loop (J) as an inside loop gives you control of the car's speed. You can experiment with various upper limits for the J loop to achieve the rate of speed desired, with the amount of distortion that you can tolerate. You can also use PAUSE 1 in place of the J loop. However, you will be trading distortion for a very slow ride with some bouncing!

Remember that scrolling can only be used on a low-resolution screen.

I hope that you have enjoyed and will find many programming uses for both scrolling and user-defined characters.

V High-Resolution Graphics

Introduction

Simons' BASIC uses two screen modes, the low-resolution mode and the graphics mode. You are familiar with the low-resolution mode whose screen mapping is 40 columns (0 through 39) by 25 rows (0 through 24), with each character cell (byte) consisting of 8 by 8 pixels (bits) allowing two colors—foreground color and background color. Only the low resolution mode has been used in the previous chapters.

Graphics Screens

The graphics mode has two screen types—high resolution and multicolor. Both screen types use the graphics commands reviewed in this chapter. The differences between these two screen types are summarized below.

In this section you will be using the high resolution screen. As the name indicates, this screen type provides the highest screen resolution possible on the Commodore 64, allowing you control over each pixel (short for picture element) on the screen—200 rows of 320 pixels each. Colors are handled in exactly the same way as on the low-resolution screen, with each character cell (byte) allowing only two colors—foreground and background. This screen type should be used when fineness of detail and smoothness of line is most important.

The multicolor screen uses the same graphic commands that the high-resolution screen uses. The difference is in the presentation on the screen because of the lower resolution—200 rows of 160 pixels. This is only half of the horizontal resolution available on the high-resolution screen. However, with multicolor you have three foreground colors available for each character cell (byte). This is the screen type to use

when multiple colors and the mixing of color is most important. This screen type sacrifices the higher resolution for the increased use of color. The multicolor screen, and the difference in the screen presentation of each of the graphic commands, is reviewed in Section VI.

High-Resolution Screen

The Simons' BASIC high resolution screen configuration multiplies the low resolution addressable screen locations by eight, addressing each bit! As you know, in low-resolution (the normal Commodore screen) each character cell uses a one-byte screen location, with each byte consisting of 8 by 8 bits. The screen is 40 bytes wide (columns) and 20 bytes deep (rows). Thus the high-resolution screen, by addressing each bit, is mapped as 32Ø columns (40*8), and 2ØØ rows (25*8). This permits addressing at the smallest addressable point on the screen, called bit-mapped addressing. Each graphics bit is called a "pixel", short for picture element. The important difference between the low-resolution and high-resolution screen is simply multiples of eight. The old screen addresses that you have been using with Commodore BASIC and Simons' low-resolution commands are multiplied by eight for the high-resolution screen. If you have been using Commodore BASIC graphics, employing PEEK, POKE, AND, OR, and NOT, all of these terms are familiar to you. What will be unfamiliar to you is the ease of creating Simons' BASIC screen graphics. The screen layout form below, Figure S5.1, shows both low-resolution and high-resolution screen mapping, providing a comparison of the two screen types. The change to a high-resolution screen configuration will not be confusing if you simply think of the screen locations in the terms you already know—low-resolution—and multiply by eight for the locations on a high resolution screen. This will give you both the column and row equivalents.

As you can see from the screen layout (S5.1), the center of a high resolution screen is 16Ø,1ØØ—column 16Ø, row 1ØØ. Column and row are referred to as the x, y coordinates in all Simons' BASIC graphic commands. The x coordinate is the column location, and the y coordinate is the row location. Remember, you now have 32Ø columns and 2ØØ rows, or, in other words, a matrix or grid 32Ø by 2ØØ pixels.

All of the commands reviewed in this chapter are graphic commands, and, therefore, are used on both the high-resolution and multicolor screens. Once you have learned to use these commands on a high-resolution screen, you will also understand how to use them on the multicolor screen.

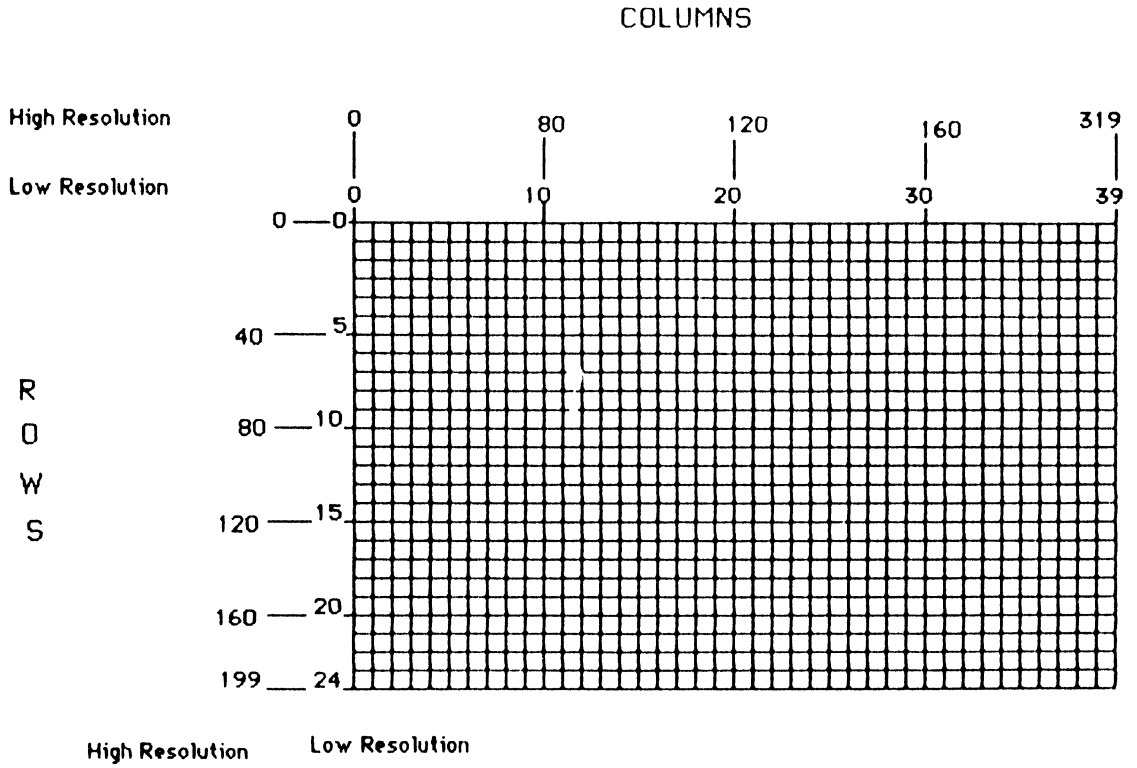


Figure S5.1: Screen Layout with Column Coordinates (Showing both high-resolution and low-resolution screen locations.)

The following is a brief overview, by chapter, of the Simons' BASIC graphic commands.

Chapter 15

HIRES is the command that initiates the bit-mapped, high-resolution screen. It also sets both the foreground and background screen colors.

REC is the command used to plot—"draw"—rectangles and squares of any size at any screen location.

TEXT is the graphic command that replaces the low-resolution PRINT command, allowing the user to print character strings on a graphic screen. Characters can be printed in both upper and lower case, in varying heights, and with varying spacing between characters.

CHAR prints a single character on a graphics screen, using the Commodore poke code. It also allows varied character height.

LOW COL allows the user to change foreground color(s) used on portions of a graphic screen.

HI COL returns the foreground color(s) used to the original color(s) specified.

LINE plots a line on a graphics screen at any screen location the user specifies.

CIRCLE plots circular and elliptical shapes at any user-specified screen location.

ARC draws a portion of the circumference of a circular shape at a user-specified location.

ANGLE draws the radius of a circle at a user-specified location.

PLOT fills—"turns on"—one pixel at a specified location.

TEST returns the status—"on" or "off"—of a user-specified screen location of one pixel.

NRM returns the Commodore 64 from a graphic screen back to the low-resolution screen type.

Chapter 16

PAINT fills any enclosed area—plotted shape—with a foreground color.

BLOCK plots a rectangle or square and fills it with a foreground color.

DRAW allows the user to create a graphic design, pixel by pixel, to be displayed at a specific screen location.

ROT, short for rotate, allows a shape created with the DRAW command to be displayed at specified angles of rotation in a user-specified size.

Chapter 17

CSET, using the parameter 2, re-displays the previous graphic screen.

COPY produces a hard copy of a graphic screen on a serial printer.

Key to Entering Program Listings

There are no additional key conventions used in this section.

15 Plotting Graphics

HIRES

This command sets the screen into high-resolution mode, a matrix of 320 by 200 pixels, where all points are plotted pixel by pixel. This command must be used with both graphic screens. For a high-resolution screen, it also sets the plotting and screen background colors. This command's function with a multicolor screen is described in Section VI.

Command format:
HIRES plotting color,background color

High-resolution mode allows two colors—one for plotting and one for screen background. The plotting color is the foreground color, the color in which the high-resolution graphics will be displayed. The background color specifies the screen color. The Commodore colors and their numbers are shown on page 108, Figure 13.1.

REC

This command draws a rectangular shape on a graphics screen. The size and the shape of the rectangle are specified in the command parameters.

Command format:
REC x,y,width,depth,plot type

All five parameters are required. Omitting any parameter will cause a SYNTAX ERROR. The parameters x and y are the coordinates of the starting location for plotting. These coordinates are used in all plotting commands. In the REC command, the x,y coordinates specify the upper left-hand corner of the rectangle. The parameter x designates the starting column, and the parameter y the starting row. Width specifies the total number of columns the rectangle will cover, that is, the length of the horizontal (top and bottom) edges. Depth specifies the total number of rows that the rectangle will cover—length of the vertical edges (sides). The last parameter, plot type, is used with all plotting commands. The three plot types for a high-resolution screen are shown below in Figure 15.1.

Plot Type	Function
Ø	Turns off plotting, (clears what has been plotted, or leaves cleared what has not been plotted).
1	Turns on plotting.
2	Reverses plot types Ø and 1. If plotting was on, plotting is cleared (turned off). If plotting was off, plotting is turned on.

Figure 15.1

Plotting cannot exceed screen size. The beginning column plus the width (x+width) cannot exceed 320, and the beginning row plus the depth (y+depth) cannot exceed 200. If either exceeds these limits, the program will crash with a BAD MODE error message. This is the message given when a command parameter is outside the range allowed.

The first demonstration program will plot a yellow rectangle located in the center of a dark grey screen having a black border. The first parameter of the COLOUR statement sets the border color (Ø), and the HIRES statement sets both the plotting color (7) and the screen background color (11). Clear computer memory and enter the following program. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
COLOUR	1Ø COLOUR Ø,11
HIRES	2Ø HIRES 7,11
REC (plot type 1)	3Ø REC 12Ø,8Ø,8Ø,4Ø,1
	4Ø GOTO 4Ø

RUN the program, and hold down the RUN/STOP key to end the run. The rectangle that was drawn on your screen is shown below in Figure 15.2.

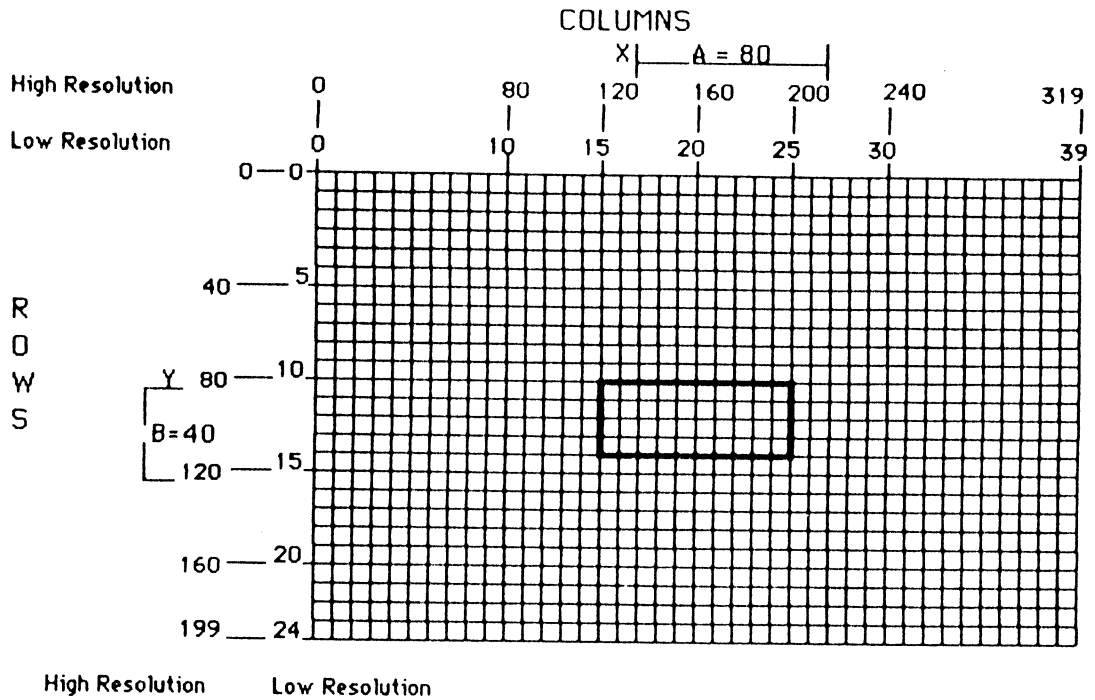


Figure 15.2: Drawing of above rectangle on a high-resolution screen.

Of course, it's unnecessary to first draw this rectangle on graph paper. Knowing that 16Ø,1ØØ are the center x,y coordinates of a high-resolution screen gives us all the information we need to position the rectangle in the center of the screen. All that is needed is to place half the rectangle on either side of these coordinates, for example, 16Ø minus half the length of the top edge (16Ø−4Ø=12Ø), and 1ØØ minus half the length of the side edge (1ØØ−20=8Ø). Just remember that the x and y coordinates designate the column and row respectively where plotting will begin.

The COLOUR statement is needed only if you want to change the standard border color. When using the high-resolution mode, the second parameter of the COLOUR statement—screen color—should be the same as the screen background color specified in the HIRES statement, or the same as the current text screen background color. If a different color is specified, the COLOUR statement will cause the screen to briefly flash that color, then change to the color specified in HIRES when that statement is performed.

To eliminate the screen border entirely, use the same border and

screen color. To do this in the program in memory, change statement line 10 to: **COLOUR 11,11**. RUN the program again and press RUN/STOP to end.

In the REC statement used in this program, the plot type parameter is 1, which turns on plotting. To demonstrate plot type 0—clearing the screen of a plotted graphic—add the following statements to the program in memory. The Simons' BASIC commands illustrated are shown to the left of the statements in which they occur.

Command Illustrated	Program Statements
	40 PAUSE 2
PAUSE	50 REC 120,80,80,40,0
REC (plot type 0)	60 PAUSE 1
PAUSE	70 GOTO 30

These statements will cause the program to pause for two seconds after plotting the rectangle, then clear the rectangle using plot type 0. By ending the program with a GOTO statement, the program will repeat—turning the plotted rectangle on and then off—causing the rectangle to appear to be flashing.

RUN the program. You now see the yellow rectangle turning on and off, with a pause of two seconds on and one second off. Hold down the RUN/STOP key to end the program.

Now add the following statements to demonstrate reverse plot type, type 2. The Simons' BASIC commands illustrated are shown to the left of the statement in which they occur.

Command Illustrated	Program Statements
	70 REC 120,80,80,40,2
REC (plot type 2)	80 PAUSE 3
PAUSE	90 GOTO 70

After being plotted (plot type 1) and turned off (plot type 0), the rectangle will be alternately turned back on, then off, by repeating line 70 which specifies plot type 2.

RUN the program. To end the program and return to the normal screen colors, press the RUN/STOP and RESTORE keys.

Plotting a Square

The REC command can also be used to plot a square. You will probably assume that this can be done easily by using the same number for

both the width and depth parameters, for example: a square of 30 columns by 30 rows. But, because your video screen is rectangular rather than square, an adjustment must be made for the ratio of screen width to screen depth. Television screens and monitors have a ratio of approximately 1.375, width to depth. On a high-resolution screen the ratio of 1.4 will be used to adjust the horizontal lines (top and bottom edges)—the width parameter of the REC command—to form a square. To demonstrate this, the following statements will plot two squares, one in the upper left-hand corner and one in the upper right-hand corner of the screen. These statements will replace the statements used for "blinking" the rectangle. The Simons' BASIC commands illustrated are shown to the left of the statement in which they occur.

Command Illustrated	Program Statements
REC (plot a square)	40 REC 0,0,30*1.4,30,1
REC (plot a square)	50 REC 278,0,30*1.4,30,1
	60 GOTO 60

RUN the program. The screen shows a rectangle plotted in the center of the screen, and two squares—one at each corner of the top of the screen. The plotting color is yellow (7), and the screen background and border are dark grey (11). Press the RUN/STOP key to end the program.

In plotting a square to end at the right edge of the screen, care must be taken not to exceed screen width (320 columns). Remember, the width specified is multiplied by 1.4. In line 50, the width specified is 42 columns (30*1.4). This means that the last column in which plotting can begin is column 278 (320-42).

To add a large square surrounding the rectangle and turn the inner rectangle off, edit the following statement lines in the program in memory.

ENTER:

```
60 REC 50,30,150*1.4,150,1
70 PAUSE 2:REC 120,80,80,40,0
80 GOTO 80
```

RUN the program, and press the RUN/STOP key to end.

If plot type 2—reverse plot type—were used in line 70, rather than 0, the effect would be different. The four corner pixels of the rectangle would be left on. To see this effect, **change plot type from 0 to 2 in line 70**, and RUN the program again. Only ending the program will remove these four turned-on pixels!

Now you can use squares as well as rectangles on your charts or

graphs, or in your educational or game programs. Of course, you'll also want to add text, either within the square or rectangle or on another part of the graphic screen.

TEXT

The TEXT command allows you to print character strings on a graphic screen. This replaces the PRINT statement used on a low--reslution screen. If a PRINT statement is used with a graphics screen, the print characters will not appear on the screen. (Printing occurs on the low-resolution screen in memory, rather than on the HIRES screen.) The characters specified in the TEXT statement can be displayed in upper or lower case, or both, and character height and spacing can be increased. All TEXT will be printed in the plotting color specified in the HIRES command.

Command format for upper case text:
TEXT x,y,"[CTRL A]character string",plot type,height, spacing

Command format for lower case text:
TEXT x,y,"[CTRL B]character string",plot type,height, spacing

CTRL A is used to specify upper case, while CTRL B is used to specify lower case. When using only upper case this parameter may be omitted, since upper case is the default. All other parameters are required, and if omitted will cause a SYNTAX ERROR. The coordinates x,y are the column and row screen locations where the text will begin. Unlike the PRINT statement, wraparound is not possible. Text is confined to the row specified. At column 32Ø, plotting of the TEXT string will end whether the entire character string has been printed or not. No error message is given if the character string exceeds the space available on the row specified. When plotting more than one line of characters, a TEXT statement must be used for each line, with each row having a location (y) of 8 pixels greater than the last for single spacing of the text. Plot type is the same for all plotting commands.

Height is specified by using multiples of the normal character size of eight pixels. The value 1 indicates normal size—a height of eight pixels. (**Note**: The value Ø produces the same result as the value 1.) A value of 2 indicates 2*8, or 16 pixels in height; a value of 3 produces characters 3* 8, or 24 pixels in height, and so on. Height is limited only by the appearance of the characters desired. CHARACTER WIDTH CANNOT BE VARIED. With the width remaining unchanged, the height simply "stretches" the characters. It's a question of how "thin" you want your characters to be! Using the parameter 8 produces *very thin* characters 64 pixels in height.

The spacing parameter determines the number of pixels to be turned off (left blank with only the background color showing) between each character. For normal spacing, this parameter is 8—seven pixels between the end of one character to the start of the next character. Again, multiples of eight are used for additional spacing, with the parameter 16 producing double spacing, the parameter 24 producing triple spacing, and so on. Any value between Ø and 255 is valid. However, values between Ø and 8 cause a tightening of the text, with the values of Ø through 4 producing illegible text! So, if your text isn't readable, check the value being used for the spacing parameter. A value of 256 or greater will cause an ILLEGAL QUANTITY ERROR.

Figure 15.3 below gives a summary of the values of the height and spacing parameters.

Character Height

Value	*Function*
1	Normal size
2	2*normal size
3	3*normal size
4	4*normal size
5	5*normal size
6	6*normal size
7	7*normal size
8	8*normal size
9 and above	illegible text

Character Spacing

*Value**	*Function*
8	Normal spacing
16	Double spacing
24	Triple spacing
32	4*normal spacing
4Ø	5*normal spacing
48	6*normal spacing
56	7*normal spacing
64	8*normal spacing
255 is the last legal value	

*—Any value between these multiples may also be used.

Figure 15.3

Clear computer memory and enter the following program for a demonstration of using text on a high-resolution screen. You may use AUTO 10,10 for automatic line numbering. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
COLOUR	10 COLOUR 2,14
HIRES	20 HIRES 0,14
TEXT	30 TEXT 115,55,"[CTRL A]COMMODORE",1,2,8
TEXT	40 TEXT 140,75,"[CTRL A]64",1,2,8
REC	50 REC 60,15,200,110,1
	60 GOTO 60

This program will produce a 200 pixel by 110 pixel rectangle beginning at the screen location 60,15, with the character strings COMMODORE and 64 printed inside the rectangle 16 pixels in height and single-spaced. The border color (red) is set in the COLOUR statement, the screen background color (blue) in both the COLOUR and HIRES statements, and the plotting color (black) in the HIRES statement. The screen locations for the three plotting statements are shown plotted on graph paper in Figure 15.4 below.

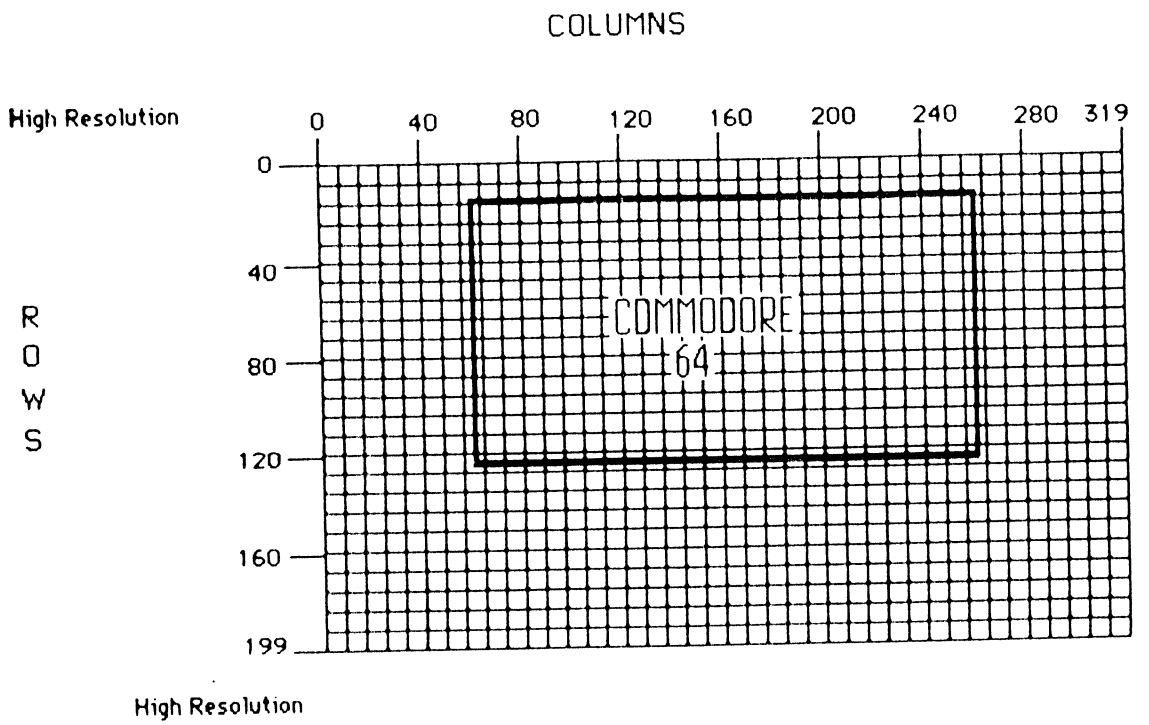


Figure 15.4: Grid showing the strings COMMODORE and 64 in a rectangle.

RUN the program and press the RUN/STOP key to end.

The TEXT statements precede the REC statement in this program, but they do not need to. Plotting statements, text or graphic, may appear in any order. The upper-case parameter used in the TEXT statements could have been omitted, since upper case is the default. The case parameter (CTRL A or B) may also be omitted when printing numbers, or either upper case or lower case may be specified. To demonstrate this, add the following statements to the program in memory.

```
35 TEXT 115,40,"COMMODORE",1,2,8
45 TEXT 140,90,"[CTRL B] 64",1,2,8
55 TEXT 140,105,"64",1,2,8
```

RUN the program. COMMODORE printed the same with and without the upper case parameter; and 64 printed the same with the upper-case parameter, the lowercase parameter, and without the parameter. Press the RUN/STOP key to end the program.

However, since you must use both the upper and lower case parameters when using mixed text—upper- and lowercase in the same character string—it is probably a good habit to regularly include this parameter. **Remove** statement lines **35, 45,** and **55,** and LIST the program. For a demonstration of mixed text, add the following program statements.

```
60 TEXT 10,150,"[CTRL A]U[CTRL B]SING [CTRL A]S
[CTRL B]IMONS' [CTRL A] BASIC",1,4,16
70 GOTO 70
```

This text will print below the rectangle, beginning at column 10, row 150, as shown in Figure 15.5. The text will print in upper and lower case, using initial caps for the words Using and Simons' and full caps for the word BASIC. The height of the characters will be 4 times normal height (32 pixels) and the characters will be double spaced.

RUN the program, noting the effects of the height and spacing parameters used in the last TEXT statement. Press the RUN/STOP key to end the program.

Both the TEXT command and another Simons' BASIC plotting command, CHAR, can be used to print a single character on a graphics screen.

CHAR

This command is more cumbersome to use than the TEXT command because the poke code for the character must be used, instead of a char-

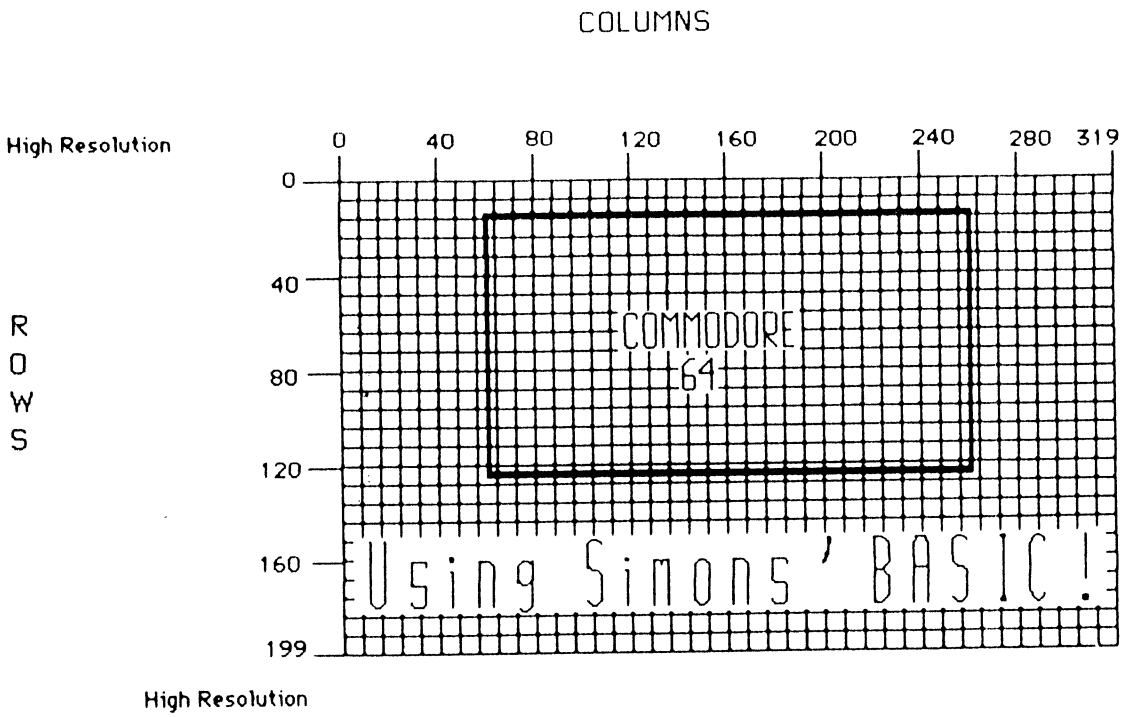


Figure 15.5: Grid showing locations of COMMODORE 64 in a rectangle and the words Using Simons' BASIC

acter string. The Commodore poke codes are shown in Appendix E, pages 132-134, of the *Commodore 64 User's Guide* that came with your Commodore 64 computer.

Command format:
CHAR x,y,poke code,plot type,height

Both commands—CHAR and TEXT—produce the same result, as shown in the next extension of the current program. As with all plotting commands, the character specified in this command will be printed in the plotting color designated in the HIRES statement. The Simons' BASIC commands illustrated are shown to the left of the program listing.

ENTER the following program statements.

Command Illustrated	Program Statements
HIRES	11 HIRES Ø,14
REC	12 REC Ø,Ø,3Ø*1.4,3Ø,1

```
CHAR
REC
TEXT
```

```
13 CHAR 15,8,3,1,2
14 REC 278,Ø,3Ø*1.4,3Ø,1
15 TEXT 287,8,"C",1,2,8
2Ø
```

The CHAR and TEXT commands (lines 13 and 15) will perform the same function in this program—each printing the character C.

The CHAR command uses the poke code for the character (3) while the TEXT command uses a character string ("C"). Note that the spacing parameter must be used in the TEXT command, even though it will not be used when plotting a single character. Failure to use this parameter will cause a SYNTAX ERROR. The HIRES command (line 11), which initiates the high-resolution screen and sets the plotting and background colors, must precede the plotting commands (CHAR, REC, and TEXT). Line 2Ø must be removed to prevent the program from initiating a new high-resolution screen. The new screen would, of course, remove the plotting performed in statement lines 12 through 15.

RUN the program. After viewing the graphic, press the RUN/STOP key to end the program. This graphics screen is shown below in Figure 15.6.

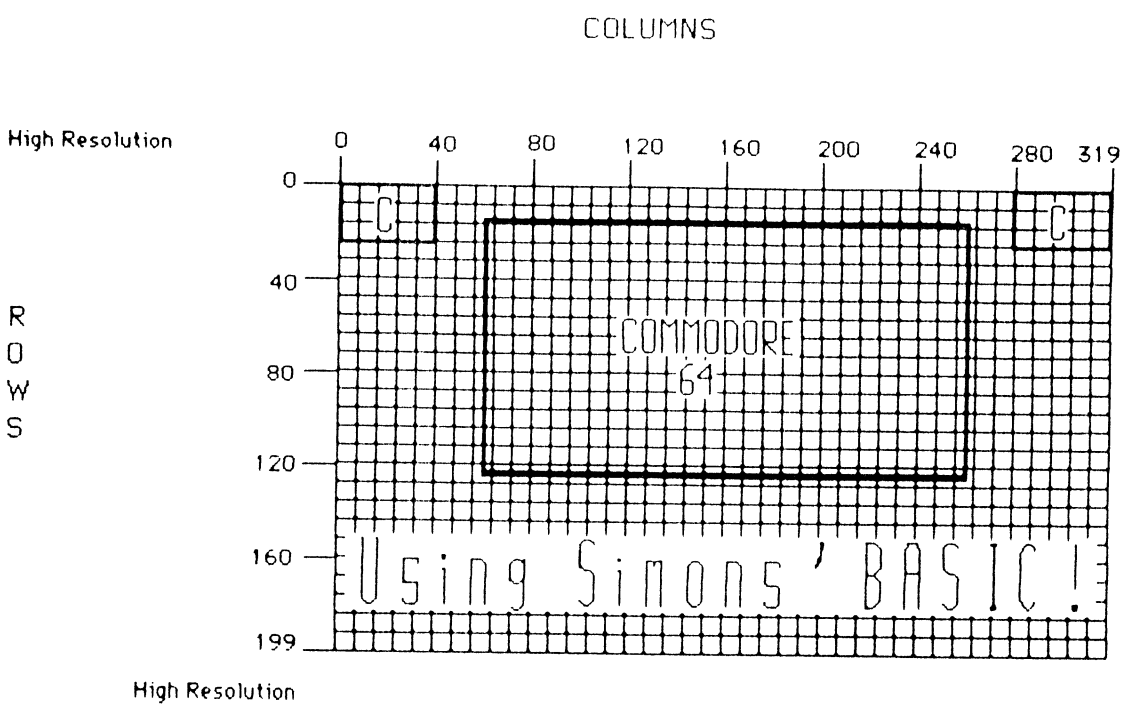


Figure 15.6: Same as Fig. 15.5 with squares containing C added.

Now we'll add the command that enables us to use additional colors in a screen graphic.

LOW COL

This command can be used both to change the plotting color originally specified in the HIRES command and to add a plotting background color. Because this command is also used to change the three plotting colors on a multicolor screen, it carries three parameters. Although the third parameter has no effect in high-resolution graphics, it **must** be specified using an arbitrary color number. If it is omitted, it will cause a SYNTAX ERROR.

Command format:
LOW COL color1,color2,color3

In high-resolution mode, color1 is the new plotting color (replacing the plotting color specified in HIRES) and color2 is the plotting background color. The plotting background color is displayed over every 8 by 8 pixel through which plotting passes. This effects all plotting commands, both graphic and text. Since the third parameter is ignored when using a high-resolution screen, you may use any valid color number for the color3 parameter.

LIST the program in memory, and add the following statement for a demonstration of LOW COL.

ENTER: `20 LOW COL 5,7,0`

After this statement is executed, the plotting color specified in the HIRES statement (black) will be replaced by the color green, and the background plotting color will be yellow, replacing that portion of the blue screen background.

RUN the program. The plotting statements that were executed prior to the LOW COL statement used the plotting color specified in the HIRES statement. All statements following the LOW COL statement used the new plotting color (green) specified in LOW COL. As you can see, the plotting background color added by LOW COL has some interesting effects! Using a different background color for the plotting than is used for the screen background causes different effects on plotted shapes, standard size text, and enlarged text. To change the plotting

color and not change the plotting background color, simply use the screen color specified in the HIRES statement as the color2 parameter in LOW COL. In this program, line 20 would be changed to LOW COL 5,14,0 to do this. You will have to experiment a bit with this parameter in order to get the special effects needed in a specific program.

Press the RUN/STOP key to end the program.

The LOW COL command may be used any number of times in a program. Adding the following statement will again change the plotting and background colors.

ENTER: `55 LOW COL 1,14,0`

When line 55 is executed plotting will be performed in white and the plotting background color and screen color will be the same—light blue.

RUN the program. After viewing the graphic, press the RUN/STOP key to end the program.

Reverting to the original plotting and background colors specified in the HIRES statement is easy with Simon's BASIC.

HI COL

This is the command used to revert to the HIRES colors if LOW COL has been used. This command changes both the plotting color and the background color back to the plotting and background colors specified in the last HIRES statement. HI COL can, of course, be followed by another LOW COL command. **Note:** Using another HIRES statement, instead of LOW COL or LOW COL and HI COL, will first clear the screen of the previous graphic. This is what makes the LOW COL and HI COL commands so important.

Command format:
HI COL

To revert back to the HIRES colors before printing the large rectangle, add the following statement to the program in memory.

ENTER: `45 HI COL`

RUN the program. The large rectangle is drawn in black on the screen background color. Plotting and plotting background color is then changed again by the LOW COL statement in line 55, causing the

character string "Using Simons' BASIC" to be printed in white. To end the program, press the RUN/STOP key.

On to the next graphic plotting tool.

LINE

This command is used for plotting a straight line anywhere on a graphics screen. Horizontal and diagonal lines may be drawn either from left to right or from right to left, and vertical lines may be drawn either downward or upward.

Command format:
LINE x,y,x1,y1,plot type

The x,y coordinates are again the beginning column and row plotting location. The parameters x1 and y1 designate the ending coordinates for plotting, with x1 designating ending column and y1 designating ending row. A negative value for any of these coordinate parameters will produce an ILLEGAL QUANTITY ERROR. However, **coordinate checking for exceeding screen size is minimal!** There is no limit checking on the x or x1 parameters (column coordinates). For example, a value of 1119 or 19ØØ for either the beginning or ending column location will be accepted, drawing a line from or to the edge of the screen. The upper limit accepted for row coordinates is 255. A value of 256 or greater will produce an ILLEGAL QUANTITY ERROR.

Plot type is the same for all graphic commands, as described at the beginning of this chapter.

In the next demonstration program, three rectangles will be plotted, located at the upper left, middle, and bottom left of the screen. A diagonal line will then be plotted from location Ø,Ø through location 319,199, which will intersect the top left and bottom right corner of each of the three rectangles. The plotting grid for this program is shown following the program listing in Figure 15.7.

Clear computer memory, and enter the following program. Use AUTO 1Ø,1Ø for automatic line numbering. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
HIRES	1Ø HIRES 7,11

```
REC
REC
REC
LINE
```

```
2Ø REC Ø,Ø,65,4Ø,1
3Ø REC 129,8Ø,65,4Ø,1
4Ø REC 254,159,65,4Ø,1
5Ø LINE Ø,Ø,319,199,1
6Ø GOTO 6Ø
```

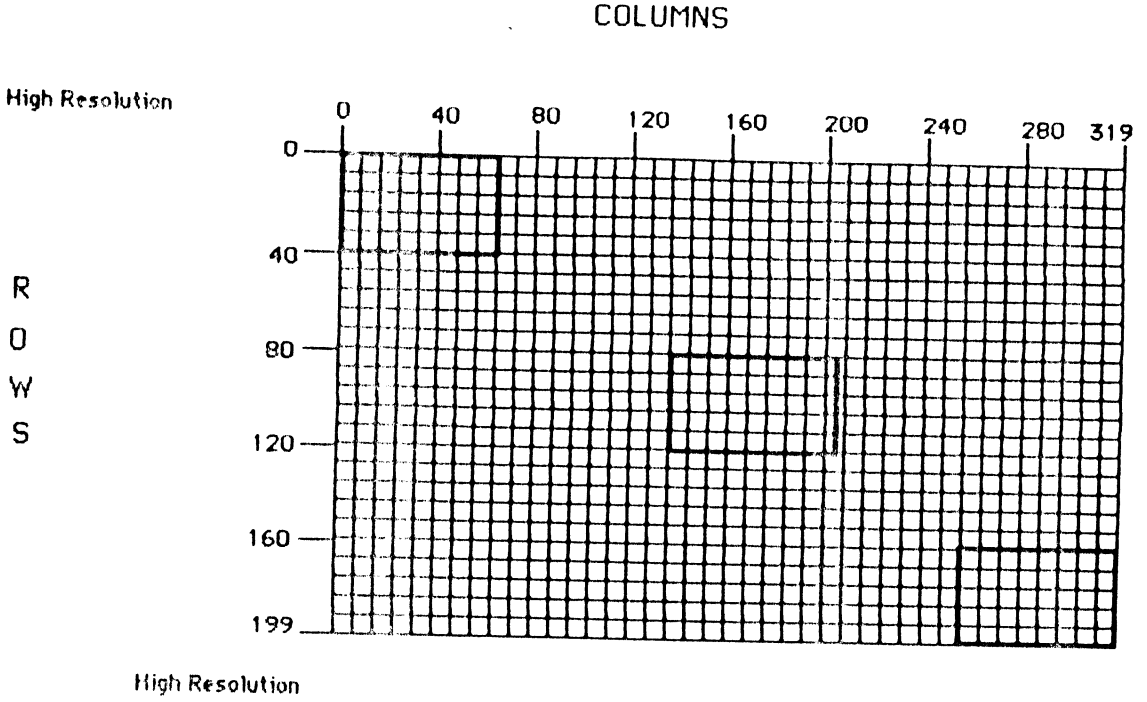


Figure 15.7: Grid showing three rectangles and a diagonal line.

RUN the program and your screen will duplicate the grid shown above. Your screen is light grey with a blue border, and the graphics are plotted in yellow. Hold down the RUN/STOP key to end the program.

Plotting a diagonal line between the opposite two corners must begin at the bottom left-hand corner and end in the upper right-hand corner. The statement needed to do this is shown below.

ENTER: `55 LINE Ø,199,319,Ø,1`

RUN the program again and press the RUN/STOP key to end.

To replace this diagonal line with a straight line bisecting the center of the middle rectangle,

ENTER: `55 LINE Ø,1ØØ,319,1ØØ,1`

and RUN the program. Hold down the RUN/STOP key to end the program.

To plot a vertical line through the center of the screen, either of the following statements can be used. The first statement will plot the line beginning at the top of the screen and ending at the bottom, and the second statement will plot the line from the bottom of the screen to the top.

ENTER:
```
55 LINE 160,0,160,200,1
```
OR
```
55 LINE 160,200,160,0,1
```

RUN the program again, using the RUN/STOP key to end the run.

Our next plotting shape will permit you to create not only circles, but interesting elliptical shapes also.

CIRCLE

This command is used to plot either a circular or elliptical shape on a graphics screen. The size of the shape, whether circular or elliptical, is set by the length specified in the horizontal radii (xr) and vertical radii (yr) parameters. Care must be taken in specifying the size of the radii so that a shape is not created that is larger than the screen size. This will cause an ILLEGAL QUANTITY ERROR.

Command format:
CIRCLE x,y,xr,yr,plot type

The first two parameters, x and y, designate the screen coordinates of the **center** of the shape to be plotted. The parameter x is the column (horizontal) coordinate, and the parameter y is the row (vertical) coordinate. The xr and yr parameters designate the horizontal and vertical radii, respectively, of the shape to be plotted. Thus their lengths determine both the size and shape of the graphic plotted. Plot type is the same for all plotting commands.

In order to plot what appears to be a perfect circle on a rectangular screen, the horizontal radii must be adjusted. This scaling factor will vary to some degree depending upon the monitor being used. Also, exact measurements will not necessarily produce the appearance of a perfect circle due to the fact that monitor screens are not a flat surface. The exact measurement for my monitor is 1.29, but a scaling factor of 1.4 produces a more perfect-looking circle! My advice is: choose a scal-

ing factor for your use that approximates the "best looking circle" on your monitor. For circles on the high-resolution screen, I will use a scaling factor of 1.4 throughout this book. That is, the horizontal radii will be 1.4 times the length of the vertical radii—horizontal radii= vertical radii * 1.4. (**Note**: When producing a hard copy of a graphics screen, however, the radii for a circle must be equal. This is covered in the review of the COPY command in Chapter 17.)

Before using CIRCLE in a program, we will experiment with this command in the immediate mode. Either DO NOT clear the program in memory or SAVE it. The program will be added to later in this chapter.

To plot a circle in the center of the screen and hold the graphic on the screen for 10 seconds,

ENTER:
```
HIRES 7,11:YR=40:XR=YR*1.4:CIRCLE 160,
100,XR,YR,1:PAUSE 10
```

A yellow circle is plotted in the center of a light grey screen. Since we did not use the COLOUR command, the border remains blue. The x,y coordinates (160,100) designate the center of the HIRES screen as the center of the circle. The vertical radii (YR) are 40 pixels in length, and the horizontal radii (XR) are 40*1.4, or 56, pixels in length. This circle is shown below in Figure 15.8, clearly demonstrating the difference in circular appearance between a printed circle with that shown on your video screen. As was noted above, to **print** a perfect circle the horizontal and vertical radii must be equal. The horizontal radii is only adjusted for viewing on the rectangular shape of a video screen. Figure 15.8 also shows both the horizontal radii (XR) and vertical radii (YR) used to configure the circle.

To elongate this circle, creating an ellipse, change the horizontal radii to: XR=(YR+YR)*1.4. These commands should now read:

```
HIRES 7,11:YR=40:XR=(YR+YR)*1.4:CIRCLE 160,100,XR,
YR,1:PAUSE 10
```

The horizontal radii are now 112 pixels—(40+40)*1.4—radiating from the center to the left and the right for a total of 224 horizontal pixels. The vertical radii remain 40 pixels each.

RE-ENTER the commands for a demonstration of an elliptical shape.

This ellipse is shown below in Figure 15.9. Again, both the horizontal and vertical radii are shown to demonstrate how the shape was drawn. And again the shape will appear wider on the graph than on your video screen.

Now a circle will be added to surround the middle rectangle of the

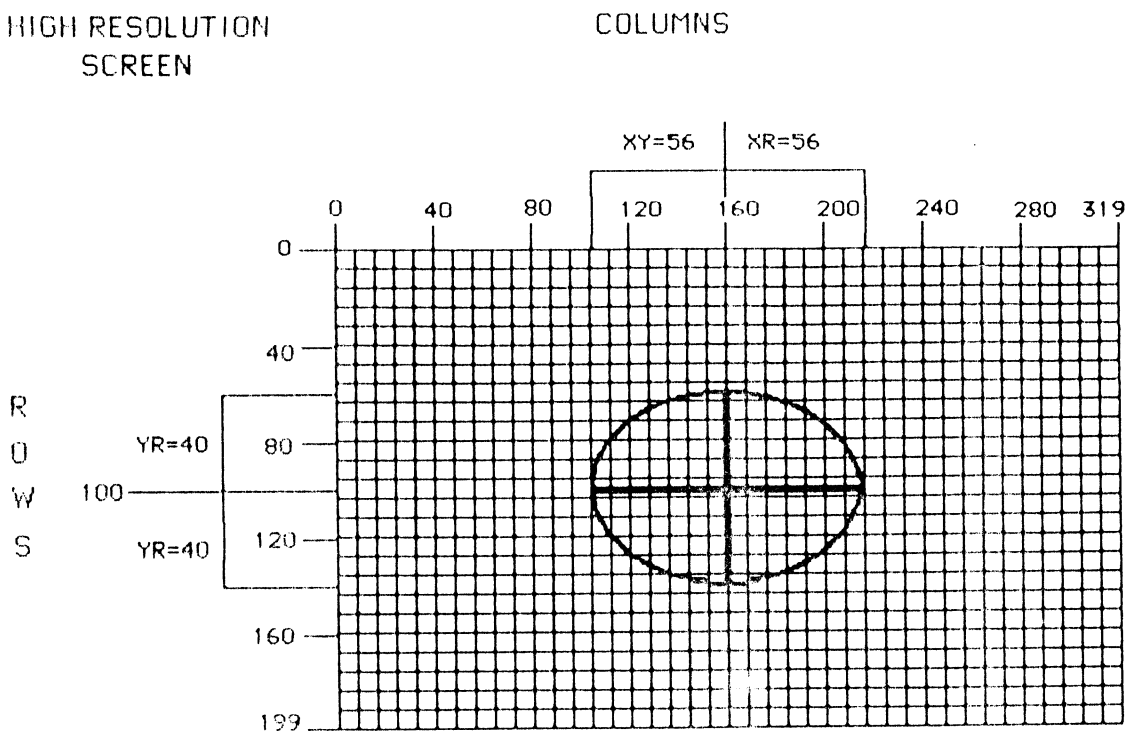


Figure 15.8: Grid showing a circle in the center.

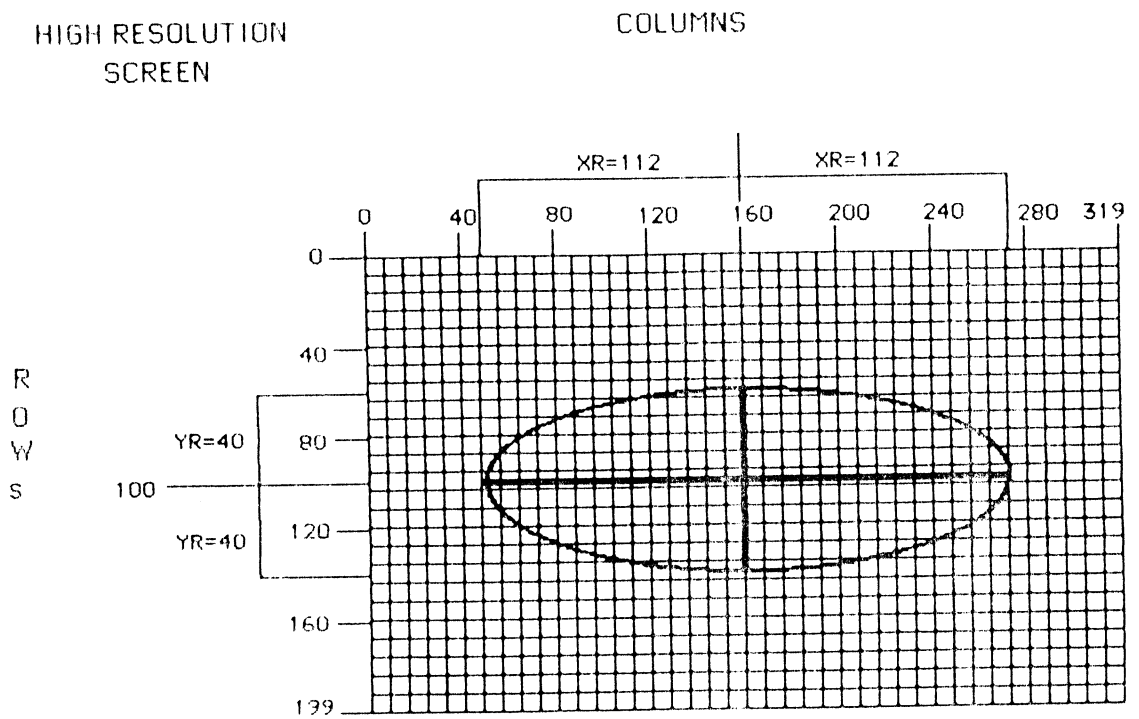


Figure 15.9: Grid showing an ellipse in the center.

graphics program in memory. In case you cleared memory and didn't SAVE the program, it is shown below for re-entry.

```
10 HIRES 7,11
20 REC 0,0,65,40,1
30 REC 129,80,65,40,1
40 REC 254,159,65,40,1
50 LINE 0,0,319,199,1
55 LINE 160,0,160,200,1
60 GOTO 60
```

If the program is still in memory, LIST the program.

ENTER: `55 YR=40:XR=YR*1.4:CIRCLE 160,100,XR,YR,1`

RUN the program. A yellow circle is plotted around the middle rectangle. Press the RUN/STOP key to end the program.

To plot only a portion of a circle, the ARC command is used.

ARC

This command plots a user-specified portion of a circular shape.

Command format:
ARC x,y,sa,ea,i,xr,yr,plot type

There are only three new parameters in this command: sa, ea, and i. The parameters x, y, xr, and yr are the same parameters used in the CIRCLE command. Plot type is the same parameter used in all plotting commands. Again, the x and y parameters are the screen coordinates for the center of the circular shape from which the arc will be drawn. The parameters xr and yr specify the horizontal and vertical radii of the circular shape of which the arc is a part. The circular shape itself need not be plotted, of course, but must be imagined and the center and radii specified in order to give the screen locations necessary for ARC plotting. The parameter sa specifies the starting degree of angle, and the parameter ea specifies the ending degree of angle. The parameter i specifies plotting as continuous, or as increments in degrees between each point on the arc. A value of 1 plots a solid arc. A value greater than 1 causes plotting to occur at intervals of the value. Specifying a degree of angle for parameter i enables a number of additional geomet-

ric shapes to be drawn. This will be demonstrated at the end of the ARC review.

Figure 15.10 shows how the parameters for this command are determined. The circle from which the arc will be taken is shown as a dotted line, while the arc is shown as a solid line. The degrees of the circumference of the circle are marked at 90 degree intervals. Figure 15.10 also shows both the horizontal radii (XR) and vertical radii (YR) used to configure the circle. Again, the arc will appear wider on the graph than on your video screen due to the adjustment made for the rectangular screen.

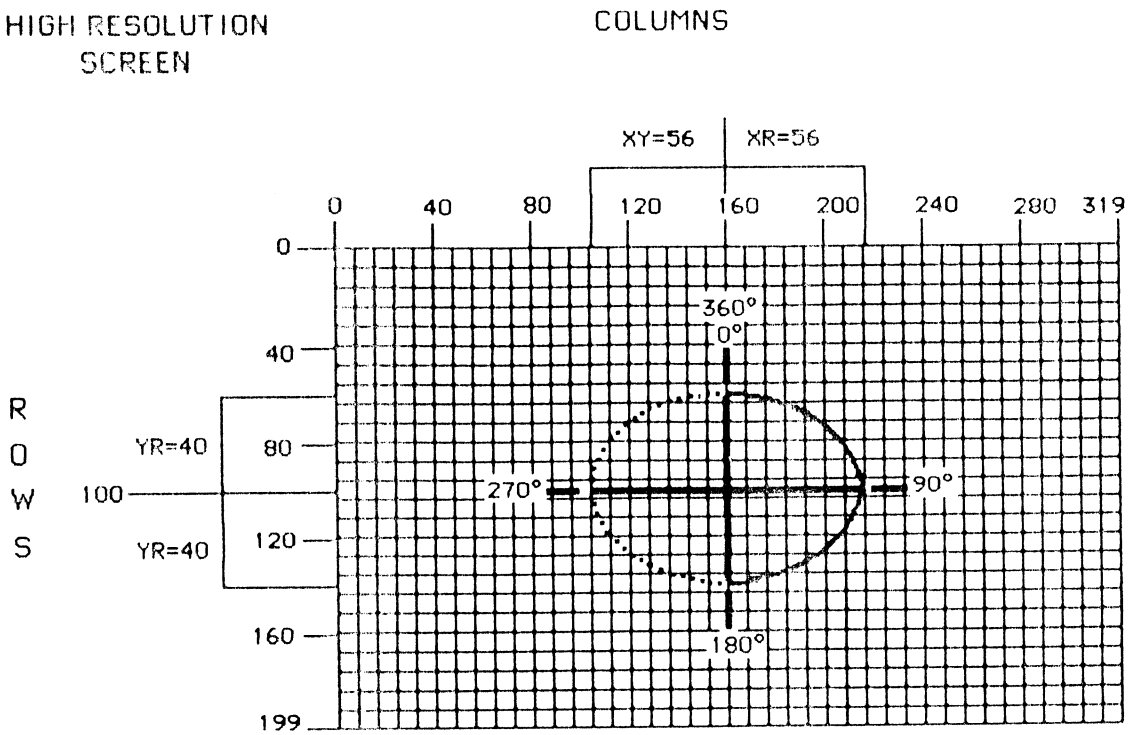


Figure 15.10: Grid showing an arc as part of a circle.

ENTER the following statements to plot both the circle and the arc shown in Figure 15.10. These statements will be added to the front of the program in memory. The Simons' BASIC commands illustrated are shown to the left of the program statements.

Command Illustrated	Program Statements
HIRES	1 HIRES 7,11
CIRCLE	2 YR=40:XR=YR*1.4:CIRCLE 160,100,XR,YR,1

```
PAUSE       3 PAUSE 3
LOW COL     4 LOW COL 8,11,Ø
ARC         5 ARC 16Ø,1ØØ,Ø,18Ø,1,XR,YR,1
            6 GOTO 6
```

The program will first plot a circle in yellow in the center of the screen, then pause for three seconds. Next, the plotting color will be changed to orange in the LOW COL statement. Then the arc will be drawn, in orange, over the right side of the circle.

RUN the program, using the RUN/STOP key to end.

Add the following statements to again pause for three seconds, then plot a second arc in blue to cover the left side of the circle.

Command Illustrated	Program Statements
PAUSE	6 PAUSE 3:LOW COL 6,11,Ø
ARC	7 ARC 16Ø,1ØØ,18Ø,36Ø,1,XR,YR,1
	8 GOTO 8

RUN the program, using the RUN/STOP key to end. **Do not** clear computer memory.

Using the immediate mode, the following commands will plot an arc—270 degrees through 90 degrees—of a circle configured: center of circle, x=80, y=40; horizontal radii of xr=20*1.4; vertical radii of yr=20. Only the arc, which is the top half of the circle, will be plotted.

```
ENTER:  HIRES 7,11:ARC 8Ø,4Ø,27Ø,9Ø,1,2Ø*1.4,
        2Ø,1:PAUSE 5
```

Remove statement lines **1 through 8**, and continue with the previous program. LIST the program. Adding the following statement lines will plot two arcs, one on each side of the circle plotted in line 55. Each arc will be located 40 pixels beyond the circle, one on each side. The Simons' BASIC commands illustrated are shown to the right of the statement in which they occur.

Command Illustrated	Program Statement
ARC	6Ø ARC 16Ø,1ØØ,5Ø,13Ø,1,65*1.4,65,1
ARC	7Ø ARC 16Ø,1ØØ,23Ø,31Ø,1,65*1.4,65,1
	8Ø GOTO 8Ø

RUN the program, using the RUN/STOP key to end. LIST the pro-

gram. To remove the distracting diagonal line from all but the two corner rectangles, replace statement line 5Ø as follows.

```
ENTER:   5Ø LINE Ø,Ø,65,4Ø,1:LINE 255,159,319,
         199,1
```

RUN the program, press the RUN/STOP key to end, and LIST the program again.

So far, you have used the number 1 as the parameter for plotting interval (i). When the number 1 is used, there is no interruption in the line of the arc that is plotted, that is, each pixel from the specified starting to ending location is plotted. When a greater number is used for this parameter, plotting will occur at that interval, separating the dots plotted. This will have the effect of causing the "jaggies", by limiting the effect of the high resolution. The "jaggies" are distortions caused by the lower resolution, having a staircase-like effect on lines that would be smooth if the number 1 were used as the plotting interval. To see this effect on the arc to the left of the circle, change the ARC parameter i (the fifth parameter) to 16 in line 7Ø. You may not want to use jagged lines often, but this parameter has another use.

Using a degree of angle for the plotting interval causes plotting to occur in a straight line from the starting angle location to the starting angle location plus the plotting interval—sa to sa+i. In other words, the plotting interval—parameter i—becomes an intermediate plotting point in the arc. In this program, changing the parameter i in the first arc (line 6Ø) to 4Ø will cause plotting to occur in a straight line from 5Ø degrees to 9Ø degrees (5Ø+4Ø), and from 9Ø degrees to 13Ø degrees (9Ø +4Ø). Likewise, in the second arc (line 7Ø) plotting will occur in a straight line from 23Ø degrees to 27Ø degrees (23Ø+4Ø), and from 27Ø degrees to 31Ø degrees (27Ø+4Ø). Edit lines 6Ø and 7Ø, replacing the value of i (now 1) with 4Ø. The two statement lines should read as follows:

```
6Ø ARC 16Ø,1ØØ,5Ø,13Ø,4Ø,65*1.4,65,1
7Ø ARC 16Ø,1ØØ,23Ø,31Ø,4Ø,65*1.4,65,1
```

RUN the program. By using a 40 degree angle for the plotting interval, both arcs were changed to a flattened version of the < and > symbols. Press the RUN/STOP key to end. This was a small example to demonstrate the difference that the plotting interval parameter can make. It can, of course, do much more!

The following program will use ARC to plot five polygons: (1) a quadrilateral (four-sided figure) on the upper left side of the screen, (2) a pentagon (five-sided figure) on the upper right side of the screen, (3) a

hexagon (six-sided figure) in the center of the screen, (4) an octagon (eight-sided figure) on the lower left side of the screen, and (5) a decagon (ten-sided figure) on the lower right side of the screen.

Clear computer memory and enter the following program. You may use AUTO 10,10 for automatic line numbering. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
HIRES	10 HIRES 14,11:Y=40:X=Y*1.4
ARC	20 ARC 60,45,0,360,360/4,X,Y,1
ARC	30 ARC 255,45,0,360,360/5,X,Y,1
ARC	40 ARC 160,100,0,360,360/6,X,Y,1
ARC	50 ARC 60,155,0,360,360/8,X,Y,1
ARC	60 ARC 255,155,0,360,360/10,X,Y,1
	70 GOTO 70

RUN the program and press the RUN/STOP key to end.

Now that you have seen how versatile this command is, you can use your imagination to create any number of shapes. This is especially useful for creating entertainment or educational software. How about writing a review program for geometry? It might also be useful for creating a special logo for a club or business association.

ANGL

This command draws the radius of a circle without having to display its circumference. Of course, any number of radii may be plotted and used in as many ways as your imagination can conceive. This is clearly shown in your *Simons' BASIC User Guide.*

Command format:
ANGL x,y,angle,xr,yr,plot type

The parameters x, y, xr, yr, and plot type are the same as described for the ARC command. Angle specifies the point on the circumference at which the radius will terminate. It is specified as a number of degrees of the circumference, that is, any number from 0 through 360. Using a clock as an example, at three o'clock the hour hand is at 90 degrees, at six o'clock it is at 180 degrees, and so on.

The following program will plot four clocks, one for each of the continental United States time zones. The hands of the clocks will be plotted by the ANGL command. By shortening the horizontal and vertical radii (parameters xr and yr respectively) the hands of the clock end inside the circle. For the radii to reach the circumference of the circle, the horizontal and vertical radii would have to be the same radii length used for CIRCLE—20*1.4 and 20 respectively. The angles needed for each hour hand are noted in statement line 30.

Clear computer memory and enter the following program. You may use AUTO 10,10 for automatic line numbering, rather than keying in the line numbers yourself. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
COLOUR	10 COLOUR 5,11
HIRES	20 HIRES 7,11:X=1.4
	30 REM ANGLES FOR THE HOUR HAND AT 1,2,3, AND 4 O'CLOCK
	40 P=30:M=60:C=90:E=120
TEXT	50 TEXT 65,30,"[CTRL A]U[CTRL B]NITED [CTRL A]S[CTRL B]TATES",1,2,16
TEXT	60 TEXT 120,60,"[CTRL A]T[CTRL B]IME [CTRL A]Z[CTRL B]ONES",1,2,8
CIRCLE and TEXT	70 CIRCLE 35,110,20*X,20,1:TEXT 10, 140, "[CTRL A]P[CTRL B]ACIFIC",1,1,8
CIRCLE and TEXT	80 CIRCLE 110,110,20*X,20,1:TEXT 85, 140, "[CTRL A]M[CTRL B]OUNTAIN",1, 1,8
CIRCLE and TEXT	90 CIRCLE 195,110,20*X,20,1:TEXT 170,140, "[CTRL A]C[CTRL B]ENTRAL", 1,1,8
CIRCLE and TEXT	100 CIRCLE 275,110,20*X,20,1:TEXT 250,140, "[CTRL A]E[CTRL B]ASTERN", 1,1,8
ANGL	110 ANGL 35,110,360,15*X,15,1:ANGL 35,110,P,10*X,10,1
ANGL	120 ANGL 110,110,360,15*X,15,1:ANGL 110,110,M,10*X,10,1

```
ANGL          130 ANGL 195,110,0,15*X,15,1:ANGL
              195,110,C,10*X,10,1
ANGL          140 ANGL 275,110,0,15*X,15,1:ANGL
              275,110,E,10*X,10,1
              150 GOTO 150
```

When the program is run, the title will be printed, then the clocks (circles) will be plotted and labeled, and last the ANGL statements will plot the clock hands. Note that either 360 degrees or 0 degrees can be used as the angle parameter for plotting the perpendicular minute hands. The first ANGL statement in lines 110 through 140 sets the angle for the perpendicular, minute hand. Lines 110 and 120 use 360 as the angle, and lines 130 and 140 use an angle of 0.

RUN the program and press the RUN/STOP key to end.

PLOT

This command displays one dot—one pixel—on a graphics screen in a user-specified location.

Command format:
PLOT x,y,plot type

The x and y parameters are, as always, the column and row coordinates specifying the location for plotting. Plot type is as previously described. All three parameters are required. This might seem like a very weak command after reviewing so many powerful plotting commands. But, like all Simons' BASIC commands, it too has its uses. You will not often plot just one pixel, but you might use this command to plot variously-spaced dotted lines. The following program demonstrates three uses of PLOT.

Clear computer memory and enter the following program. Use AUTO 10,10 for automatic line numbering, rather than entering the line numbers shown with the program. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
HIRES	`10 HIRES 0,1`

```
TEXT        20 TEXT 0,0,"DEMONSTRATIONS OF THE PLOT
            COMMAND",1,2,8
            30 REM PLOT A DIAGONAL DOTTED LINE
            40 FOR I=1 TO 80 STEP 10
PLOT        50 PLOT I+10,100+R,1
            60 R=R+8
            70 NEXT I
PAUSE       80 PAUSE 3
PLOT        90 PLOT 30,31,1
TEXT        100 TEXT 40,25,"[CTRL A]PLOT [CTRL B]IS
            USEFUL FOR HIGHLIGHTING",1,1,8
TEXT        110 TEXT 40,33,"[CTRL B]INDENTED TEXT.",
            1,1,8
PAUSE       120 PAUSE 3
            130 FOR I=0 TO 360 STEP 5
PLOT        140 PLOT I,60,1
            150 NEXT I
TEXT        160 TEXT 125,80,"INSTRUCTIONS",1,1,8
PAUSE       170 PAUSE 3
REC         180 REC 10,100,80,60,1
PLOT        190 PLOT 125,106,1:PLOT 130,106,1
TEXT        200 TEXT 140,100,"[CTRL A]F[CTRL B]OLD
            THE PAPER ALONG",1,1,8
TEXT        210 TEXT 140,108,"[CTRL B]THE DOTTED
            LINE.",1,1,8
            220 GOTO 220
```

RUN the program. Both a single dot and a double dot are used for indented text, which would be useful in printed listings. A dotted line is used to separate screen sections. A diagonal dotted line is used as an instruction guide on an illustration. Press the RUN/STOP key to end.

Since this program will be edited to demonstrate the next command reviewed, **do not** clear computer memory.

TEST

This command allows you to check for plotting at a specified screen location. A value of 0 is returned if the pixel in that location is off (no plotting) and a value of 1 if the pixel is on.

Command format:
numeric variable name=TEST(x,y)

The parameters x and y are the column and row coordinates of the location you wish to test for the occurrence of plotting. They must be enclosed in parentheses and separated by a comma. Omitting the parentheses will cause a SYNTAX ERROR.

The following program statements will be added to the front of the program in memory. A pixel of a plotted rectangle will be tested, as well as a pixel in which no plotting occurs. The Simons' BASIC commands illustrated are shown to the left of the program statements in which they occur.

Command Illustrated	Program Statements
HIRES	1 HIRES 4,7
REC	2 REC 12Ø,8Ø,8Ø,4Ø,1
TEST	3 X=TEST(16Ø,8Ø):Y=TEST(Ø,1Ø)
PAUSE	4 PAUSE 2
	5 PRINT "[CLEAR][DOWN 2] X=";X,"Y=";Y
	6 END

The rectangle plotted will travel through the location column 16Ø on row 8Ø, giving X the value of 1. Since nothing will be plotted at location Ø,1Ø, the value of Y will be Ø. So that the values of X and Y can be printed, the screen will be cleared to return to a low-resolution screen. RUN the program.

Remove statement **lines 1 through 6**, and we will add a further demonstration of TEST to the program in memory.

Enter the following program statements, using AUTO 22Ø,1Ø if you wish to use automatic line numbering. The Simons' BASIC commands illustrated are shown to the left of the statements in which they occur.

Command Illustrated	Program Statements
PAUSE	22Ø PAUSE 2
	23Ø FOR I=1 TO 2ØØ
TEST	24Ø IF TEST(1Ø5,2ØØ-I)=1 GOTO 27Ø
PLOT	25Ø PLOT 1Ø5,2ØØ-I,1

```
                260 NEXT I
PAUSE           270 PAUSE 5
```

These new instructions will first pause the graphic screen for two seconds, then draw a solid line beginning at the bottom of the screen and stopping when the dotted line, plotted earlier in the program, is reached. After a 5-second pause, the program will end.

RUN the program.

Using the PLOT command to draw a line between previously plotted locations has the advantage of being able to use TEST to determine the ending location, an advantage not available with the LINE command.

NRM

This command stands for "normal" and is used to return to the normal Commodore screen—the low-resolution screen.

Command format:
NRM

Adding this statement to the program in memory would not affect the program, nor change the display after the program ends.

16 Coloring and Drawing in HIRES

The PAINT and BLOCK commands enable you to fill plotted shapes with color. The DRAW command will allow you to create your own designs. The ROT command will enable you to display your designs in the size created, or a larger size, and position them as drawn or rotate them. ROT provides some animation, without using sprites. Each of these commands will be reviewed and illustrated in demonstration programs.

PAINT

This command is used to fill any enclosed graphic shape with the current plotting color, as specified in either HIRES, LOW COL, or HI COL. The shape must be completely enclosed. If it is not, painting takes place over the entire screen.

Command format:
PAINT x,y,plot type

The parameters x and y are the column and row coordinates that specify **any point within** the boundaries of the shape to be painted. Plot type is the same for all graphic commands. When using the high-resolution mode a shape may be painted only once. In the multicolor mode, a shape may be repainted as often as you like. This will be reviewed in Section VI—Multicolor Graphics.

To demonstrate the use of the PAINT command, the next demonstration program will:

1. Plot a rectangle then a circle. Then divide the rectangle with a diagonal line. The plotting color is black.
2. After pausing for five seconds, PAINT the lower triangular portion of the rectangle, using the diagonal line as the end of the enclosed shape. Then paint the circle. The plotting color will be purple, as specified in the LOW COL statement.
3. The plotting color will change again, this time to blue, and a hexagon will then be plotted and painted.

Clear computer memory, use AUTO 10,10 for automatic line numbering, and enter the following program. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
HIRES	10 HIRES 0,7
REC	20 REC 120,60,80,40,1
CIRCLE	30 CIRCLE 50,50,20*1.4,20,1
LINE	40 LINE 120,60,200,100,1
PAUSE	50 PAUSE 5
LOW COL	60 LOW COL 4,7,1
PAINT	70 PAINT 130,90,1
PAINT	80 PAINT 60,60,1
LOW COL	90 LOW COL 6,7,1
ARC	100 ARC 240,150,0,360,60,40*
	1.4,40,1
PAINT	110 PAINT 240,150,1
	120 GOTO 120

RUN the program. Note that only the bottom portion of the divided rectangle is painted—stopped by the diagonal line. If a location within the upper portion of the rectangle were given in the PAINT statement, then only that portion would be painted. Press the RUN/STOP key to end the program. To **also** paint the upper portion of the rectangle, add the following statement.

```
75 PAINT 190,70,1
```

To PAINT **only** the upper portion of the rectangle, use the above statement to replace line 70.

When using this command, remember that the location specified for PAINT can be any location within the plotted boundaries of the shape to be painted.

Adding the following statements to the program in memory will further demonstrate the PAINT command. For automatic line numbering when entering the following statement lines, use AUTO 12Ø,1Ø. The Simons' BASIC commands illustrated are shown to the left of the program statements in which they occur.

Command Illustrated	Program Statements
HIRES	12Ø HIRES Ø,7:X=15*1.4
TEXT	13Ø TEXT 3Ø,2Ø,"HAPPY BIRTHDAY",1,4,16
CIRCLE	14Ø CIRCLE 1ØØ,1ØØ,X,15,1
CIRCLE	15Ø CIRCLE 14Ø,8Ø,X,15,1
CIRCLE	16Ø CIRCLE 18Ø,1ØØ,X,15,1
LINE	17Ø LINE 1Ø5,115,14Ø,199,1
LINE	18Ø LINE 14Ø,95,14Ø,199,1
LINE	19Ø LINE 175,115,14Ø,199,1
LOW COL	2ØØ LOW COL 4,7,Ø
PAINT	21Ø PAINT 1ØØ,1ØØ,1
LOW COL	22Ø LOW COL 5,7,Ø
PAINT	23Ø PAINT 14Ø,8Ø,1
LOW COL	24Ø LOW COL 6,7,Ø
PAINT	25Ø PAINT 18Ø,1ØØ,1
	26Ø GOTO 26Ø

Since this added portion of the program uses locations used earlier in the program, another graphic screen must be created. Thus a second HIRES statement is needed to create this **new** graphic screen (line 12Ø).

RUN the program and press the RUN/STOP key to end.

BLOCK

This command performs the functions of the REC and PAINT commands combined. To fill an **entire** rectangle or square with the most recent plotting color specified, use BLOCK. The rectangle or square will be plotted and filled simultaneously. Any shape other than a rectangle or a square must use the command for the desired shape and the PAINT command.

Command format:
BLOCK x,y,x1,y1,plot type

The x and y parameters specify the location of the upper left-hand corner of the rectangle or square. The parameters x1 and y1 specify the bottom right-hand corner of the rectangle or square. The parameter x always specifies the column and y the row. Plot type is the same for all graphic commands.

The following demonstration program illustrates the use of BLOCK in plotting two rectangles and a square filled with the most recently specified plotting color.

Clear computer memory and enter the following program. To use automatic line numbering enter AUTO 10,10. The Simons' BASIC commands illustrated are listed to the left of the program listing.

Command Illustrated	Program Listing
HIRES	10 HIRES 4,7
BLOCK	20 BLOCK 10,10,90,50,1
LOW COL	30 LOW COL 5,7,0
BLOCK	40 BLOCK 130,70,80*1.4+80,110,1
LOW COL	50 LOW COL 6,7,0
BLOCK	60 BLOCK 230,150,310,190,1
PAUSE	70 PAUSE 3

The BLOCK statement in line 40 has to carry the adjustment for plotting a square. RUN the program.

DRAW

This command is used to designate a design to be displayed on a graphic screen.

Command format:
DRAW "design instructions",x,y,plot type

The last three parameters are already familiar. The x,y parameters (column,row) specify the location where drawing will begin. Plot type is the same for all graphic commands. The new parameter is design instructions. They may be included directly in the DRAW command as long as the statement line does not exceed 80 characters. For larger

designs the instructions are assigned as the value of a string variable, and the string variable name used as the parameter in the DRAW command. The string must, of course, be created prior to executing the DRAW command. Design instructions must always be enclosed in quotation marks. Each design is limited to 255 instructions, with no more than 74 instructions on a program line. Use concatenation to link your design instructions when they exceed 74. For example: A$= "555666776785 etc.":B$=A$+"885567789". Each instruction is designated by a number, as shown in Figure 16.1, specifying the location of the pixel to be turned on—plotted—or left off—skipped over. This provides the ability to leave spaces between dots drawn while continuing the same graphic. You should use these instructions as you would move a pencil on paper.

DRAW Command Design Instructions

Number	*Instruction*
0	Move one pixel to the right.
1	Move one pixel up.
2	Move one pixel down.
3	Move one pixel to the left.
5	Move one pixel to the right and plot a dot.
6	Move one pixel up and plot a dot.
7	Move one pixel down and plot a dot.
8	Move one pixel to the left and plot a dot.
9	Stop drawing.

Figure 16.1

Instructions 0 through 3 do no plotting; they simply move the "pencil". Instructions 5 through 8 move the pencil **and** plot a dot (pixel). Instruction 9 is only needed to end "drawing" before the end of the design instructions. This would be useful for modifying a design. Inserting the instruction 9 at the point where the modified design should end would save the additional design instructions, which might be useful at another time. Normally the end quotation mark of the last design statement signifies the end of a design.

Since each instruction affects only one pixel (bit), an enormous number of instructions would be required for a large graphic if the size of the design could not be increased. Fortunately, automatic design

enlargement is provided in a companion instruction, the ROT—standing for rotate—command. The ROT command **must** be used with the DRAW command in order to display a design.

ROT

This command is used to display the design specified by the DRAW command. It also provides design rotation and design enlargement.

Command format:
ROT rotation number,size

The parameter rotation number designates the degree of rotation desired. Rotation can be eliminated by specifying the value ∅ as the rotation number. This means that the design will be shown at the angle at which it was drawn in the design instructions. Rotation numbers and the degree of rotation that each represents are shown in Figure 16.2 below.

ROT Rotation Instructions

Rotation Number	*Degree of Rotation*
∅	0
1	45
2	90
3	135
4	180
5	225
6	270
7	315

Figure 16.2

A rotation number greater than 7 will cause the error message BAD MODE. A negative number will cause an ILLEGAL QUANTITY ERROR.

The parameter size specifies the desired size of the design. The number 1 is used to display the design at the normal size, that is, as drawn.

Each number higher than 1 increases the size by that amount. For example, the number 2 would double the size of the design, while 3 would triple the size, and so on. If the size specified creates a design larger than screen size, no error message will be given. The portion of the design that falls outside of the screen will simply not be seen. A value of Ø will not generate the correct design. Using the following program as an example, the design would be a vertical and horizontal line, rather then a pendulum.

The following demonstration program will show a design in size 1—**as drawn**—to illustrate the size of a pixel-by-pixel drawing. It will be shown at Ø degree of angle, that is, as drawn with **no change of position**. The design object is a pendulum, consisting of a straight rod ending with a square pendulum bob (the weight).

Clear computer memory, use AUTO 1Ø,1Ø for automatic line numbering, and enter the following program. To aid you in keying in the design string in line 2Ø, here are simplified keyboarding instructions: twelve 7's, two 5's, three 7's, four 8's, three 6's, and three 5's. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
HIRES	1Ø HIRES 7,Ø
	2Ø A$="777777777775577788886665555"
ROT	3Ø ROT Ø,1
DRAW	4Ø DRAW A$,15Ø,7Ø,1
PAUSE	5Ø PAUSE 5

RUN the program.

Edit line 3Ø, as follows, to increase the pendulum to four times its normal size.

```
3Ø ROT Ø,4
```

ROT now specifies Ø rotation (no rotation) and an increase of 4 in the size of the design.

RUN the program again. The size is now large enough to clearly see the design. As I'm sure you noticed, the horizontal lines were deeper in color than the vertical lines. This difference disappears when using black as the plotting color. To effect lines of equivalent intensity, switch the plotting and background colors in the HIRES statement (line 1Ø). This statement will now read: **HIRES Ø,7**.

RUN the program again.

To demonstrate size increases, the following program changes will display the design in sizes 1 through 7. The Simons' BASIC commands illustrated are shown to the left of the program statements in which they occur. LIST the current program, and **enter or edit** the statements as shown below.

Command Illustrated	Program Statements
	25 FOR I=1 TO 7
ROT	30 ROT Ø,I
PAUSE	50 PAUSE 2
DRAW	60 DRAW A$,150,70,Ø
	70 NEXT I

Each design will be turned off (line 60) before displaying the design in the next size.

RUN the program.

It is often useful to use the ROT command only to increase the size of a design, and to bypass rotation by using the parameter Ø.

For a demonstration of the rotation parameter, **clear computer memory**, enter AUTO 10,10 for automatic line numbering, and enter the following program. This program will use the same design, redrawn to assume a 45 degree angle. This was necessary in order to correctly position the pendulum for rotation. For demonstration purposes, this rotation is exaggerated. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
HIRES	10 HIRES Ø,7
	20 A$="555555555555556655577778886"
	30 FOR I=Ø TO 4
ROT and DRAW	40 ROT I,3:DRAW A$,150,100,1
DRAW	50 DRAW A$,150,100,Ø
	60 NEXT I
	70 FOR J=4 TO Ø STEP -1
ROT and DRAW	80 ROT J,3:DRAW A$,150,100,1
DRAW	90 DRAW A$,150,100,Ø
	100 NEXT J
	110 GOTO 30

Loop I will rotate the pendulum from its beginning angle of 90 degrees to an angle of 180 degrees, using the 45 degree increments

imposed by the rotation parameter. Loop J will swing the pendulum back by rotating the pendulum from the 180 degree position to the beginning 90 degree position, again using the 45 degree increments. RUN the program and press the RUN/STOP key to end.

The PAUSE command can be used to slow rotation. But even a pause of one second is greater than desired. To see the result of a one second pause between rotations, add the following statements to the program in memory.

```
45 PAUSE 1
85 PAUSE 1
```

RUN the program again and press the RUN/STOP key to end. You can substitute a FOR NEXT loop for each of the above PAUSE statements. FOR K=1 TO 50:NEXT K and FOR L=1 TO 50:NEXT L works nicely.

A design need be only a portion of a graphic screen on which other graphic commands are also utilized. This is demonstrated by turning the program in memory into GRANDDAD'S CLOCK. Add the following statement lines to the program in memory. The Simons' BASIC commands illustrated are shown to the left of the program line in which they occur.

Command Illustrated	Program Listing
REC	12 REC 110,10,80,150,1
REC	13 REC 115,99,70,40,1
CIRCLE	14 CIRCLE 150,45,20*1.4,20,1
ANGL	15 ANGL 150,45,90,15*1.4,15,1
ANGL	16 ANGL 150,45,0,15*1.4,15,1
TEXT	17 TEXT 90,180,"GRANDDAD'S CLOCK",1,2,8

Now make the following changes to the program.

1. Change the **size** parameter in **both** ROT statements (lines 40 and 80) **to 1**. In order to have a reasonable size clock, the size of the pendulum must be decreased.
2. Increase the length of the pendulum rod by **adding eight number 5 instructions** to the beginning of the A$ design character string (line 20).
3. **Remove** lines **45** and **85**, the PAUSE or FOR NEXT statements that were added to slow the pendulum.

RUN the new graphic program. Press the RUN/STOP key to end.

If you are not going on to the next chapter now, SAVE this program. It will be used in Chapter 17 to demonstrate output commands.

17 Graphic Output

The two output commands used for graphic screens are CSET and COPY. As you know, CSET has two functions that are used on a low-resolution screen, using the parameters Ø and 1. The third function of CSET, using 2 as the parameter, is to recall a previously displayed graphic screen. The COPY command permits you to produce a hard copy of a graphic screen on a serial printer. COPY is the same for graphic screens as HRDCPY (Chapter 9) is for low-resolution screens.

CSET

As with the other two functions of CSET (Ø and 1), CSET 2 can also be used in both the immediate and program modes.

Command format:
CSET 2

CSET 2 is used **after** the program has left the high-resolution or multi-color mode and returned to the low-resolution mode (the normal screen type). Used in the immediate mode after the program ends and returns to a low-resolution screen, CSET 2 must be followed by a command, or group of commands, to hold the graphic on the screen. PAUSE is the most useful command for this purpose. To recall a screen from the program mode, the program must be using a low-resolution screen at the time of recall. This means that if the program has been using high-resolution, it must be switched to low-resolution before using CSET 2. This is done by first using CSET Ø or CSET 1 to return—even briefly—to a low-resolution screen.

The use of CSET 2 in both the immediate and program modes will be

demonstrated using the program from Chapter 16. If you cleared memory at the end of the last chapter, LOAD this program—GRANDDAD'S CLOCK—now.

RUN the program and press the RUN/STOP key to end. Ending the program will bring you back to the low-resolution mode.

ENTER:
```
CSET 2:PAUSE 10
```

The graphic screen reappeared. The pendulum may or may not have been displayed. It would not be displayed if either line 50 or line 90 was the last statement executed when the program was terminated.

LIST the program. To demonstrate screen recall in the program mode, the program will be changed to replace the endless GOTO loop with a FOR NEXT loop and the recall instructions will be added.

ENTER:
```
25 FOR K=1 TO 10
110 NEXT K
120 CSET 0:PRINT "[CLEAR]":PAUSE 3
130 CSET 2:PAUSE 5
```

The above program changes will cause the pendulum rotations (loops I and J) to be repeated 10 times. Then the screen will return to low-resolution mode and be cleared (line 120). After a PAUSE of 3 seconds (line 120), the graphic screen will reappear and PAUSE for 5 seconds (line 130) before the program ends, returning again to the low-resolution screen. RUN the program.

It was not necessary to clear the screen or pause before recalling the graphic screen. It is only necessary to first return to a low-resolution screen by using either CSET 0 or CSET 1. PAUSE was used in this program to demonstrate the switch from high to low resolution. If the CLEAR statement wasn't used, the most previous low-resolution screen (in this case, the program listing) would be displayed.

COPY

This statement outputs the contents of a graphic screen to a serial printer.

Statement format:
COPY

To print the program in memory, the CIRCLE and ANGL statements must be changed so that the x and y radii are equal. If you have a serial printer, make the necessary statement changes and add the copy statement as shown below.

ENTER:
```
14 CIRCLE 150,45,20,20,1
15 ANGL 150,45,90,15,15,1
16 ANGL 150,45,0,15,15,1
140 COPY
```

Remember that the adjustment factor must always be removed from graphic statements in order to print hard copy graphics having the correct proportions. The adjustment factor is only necessary for screen display.

RUN the program to produce hard copy output. **Note: Printing success is very slow!** It will take some time for the graphic screen to print. If this task is too hard on your printer, end it. It's also possible that this command won't work with your particular printer interface. So if COPY doesn't work, the Simons' BASIC cartridge isn't the problem.

After printing the graphic screen, **edit** statements **14, 15, and 16** to again correctly display the clockface and hands on your screen.

VI Multicolor Graphics

Introduction

Multicolor is the third, and final, screen mode provided by Simons' BASIC. As you know, the multicolor screen is the second type of graphic screen available in Simons' BASIC. It also uses all of the graphic commands reviewed in the previous chapter. As you will see, the difference in output is twofold. Multicolor allows far more extensive use of color, but sacrifices the extent of resolution. The horizontal resolution for multicolor is half that of high-resolution, with each plotting position being two pixels wide. In this section you will discover the advantages and disadvantages of the multicolor screen, as compared to the high-resolution screen. You will then know which graphic screen is most apt to suit a particular programming purpose.

Multicolor Screen Configuration

The multicolor screen configuration is 160 columns wide by 200 rows deep, making each plotting position two pixels wide and one pixel deep. The y (row) coordinates are the same as for the high-resolution screen, but the x (column) coordinates change. The grid below shows the three screen configurations—low-resolution, high-resolution, and multicolor—for your comparison.

As shown on the following grid, the center of a multicolor screen is 80,100, that is, column 80 and row 100. Already being familiar with the high-resolution screen, you only have to remember that you have half the number of HIRES horizontal plotting positions available on a multicolor screen.

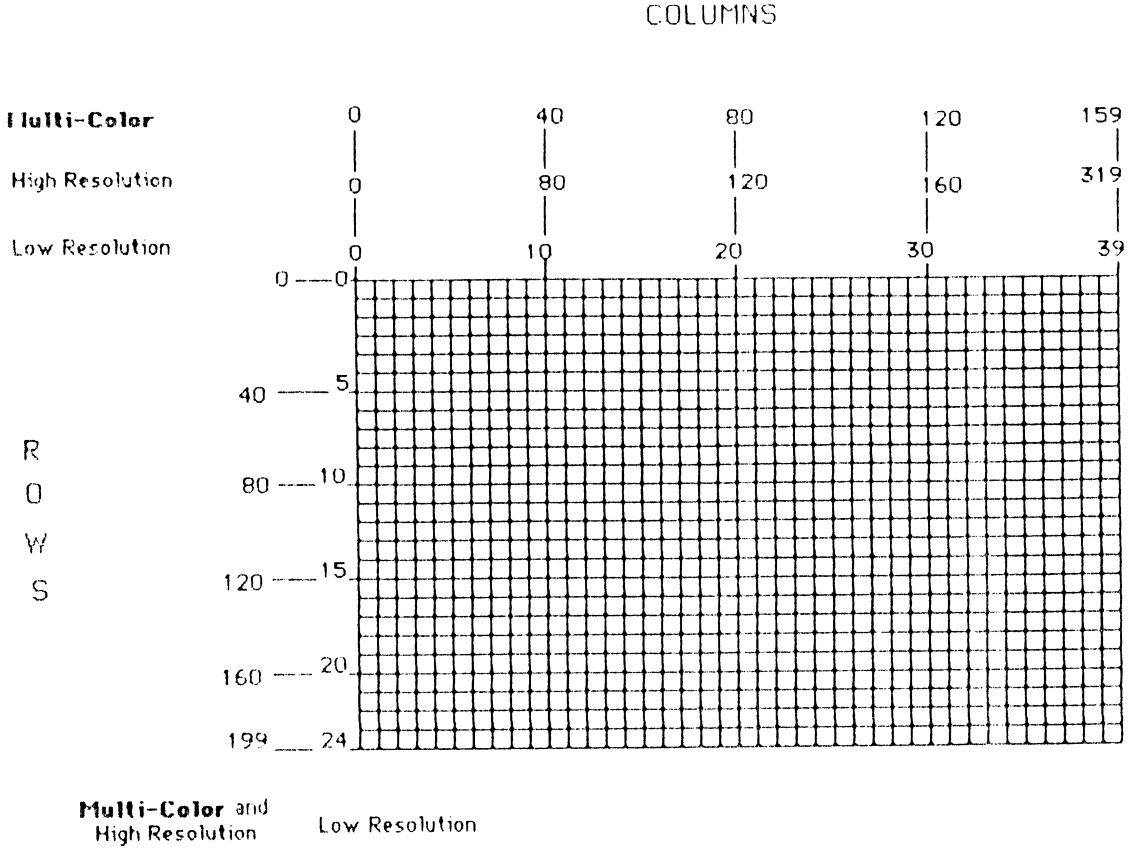


Figure S6.1: Screen Layout with Column and Row Locations (Showing multicolor, high-resolution, and low-resolution screen locations.)

There is only one graphic command specific to this screen type—the MULTI command. And, because of the increased plotting colors available, the plot type functions increase. The six graphic commands—REC (when plotting a square),CIRCLE, ARC, ANGL, LOW COL, and PAINT—are used somewhat differently on a multicolor screen. The required screen adjustment made for plotting commands is added to the depth rather than the width parameter. The LOW COL command is now able to use the color 3 parameter, and PAINT can be used to refill shapes with different colors. These commands and parameters will be re-reviewed using a multicolor screen, and the other graphic commands previously reviewed will be demonstrated with them.

The following are highlights of this section's Chapter 18.

Chapter 18

MULTI is the command that initializes the multicolor screen and specifies the three plotting colors to be used.

Plot type for the multicolor screen includes five functions: clearing

(turning off) a dot, plotting in one of three colors, and inversing dot color.

REC applies the screen adjustment to the depth, rather than to the width, parameter to plot a square on the multicolor screen.

CIRCLE, ARC, and ANGL apply the screen adjustment to the vertical radii, rather than the horizontal radii, when using the multicolor screen.

LOW COL is used to change the three multicolor plotting colors.

PAINT fills a shape with a different color as often as you like.

Key to Entering Program Listings

There are no additional key conventions used in this section.

18 Multicolor Plotting

The MULTI command with its five plot types will be demonstrated in program mode, along with the plotting commands used with the high-resolution screen. Special emphasis will be placed on those plotting commands whose parameters are modified by the multicolor screen configuration or the use of extended color. This screen configuration, which was shown in the section introduction, is shown again in Figure 18.1 below.

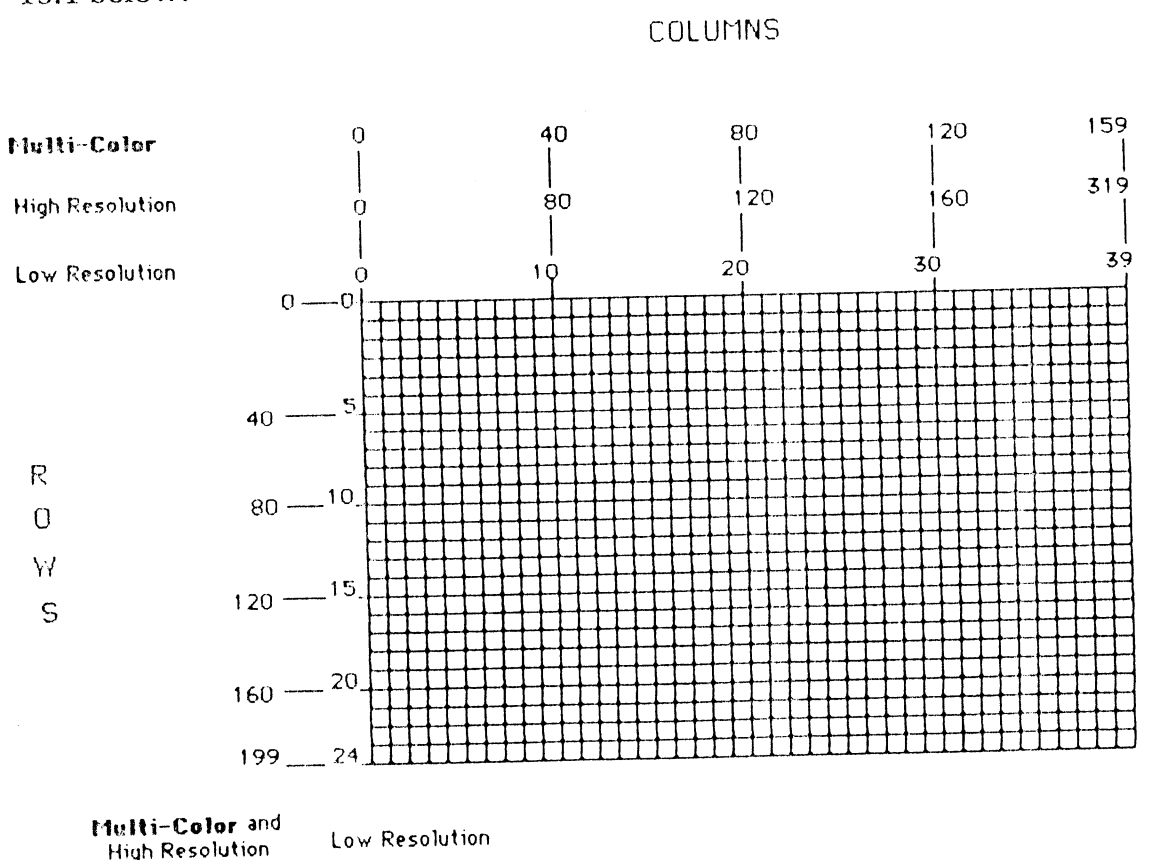


Figure 18.1: Screen Configuration Grid for the multicolor Screen

MULTI

This command resets the high-resolution screen into the multicolor screen configuration of 160 columns by 200 rows, changing each plotting position to two pixels in width. It carries the three parameters that allow three plotting colors to be used on this graphic screen.

Command format:
MULTI color 1, color 2, color 3

This command **must** be preceded by the HIRES command in order to function. MULTI, used alone, cannot initialize a multicolor screen. The HIRES command must be entered in the same format with MULTI as it is entered to initialize a high-resolution screen. However, the two **HIRES parameters**, specifying plotting color and screen background color, **are ignored**. Only the plotting colors in the MULTI command are used. The plotting color in the HIRES statement is overridden by the plotting color parameters in MULTI. The screen background color is also ignored. Therefore, the COLOUR command must be used to specify a screen background color for a multicolor screen.

Plot types and their use with the three color parameters are shown below in Figure 18.2.

Plot Type	*Function*
0	Clears a dot.
1	Uses plotting color specified in the color 1 parameter.
2	Uses plotting color specified in the color 2 parameter.
3	Uses plotting color specified in the color 3 parameter.
4	Reverses the order of the plotting color used: Plot color 0 changes to color 3 Plot color 1 changes to color 2 Plot color 2 changes to color 1 Plot color 3 changes to color 0

Figure 18.2

The above five plot type functions for a multicolor screen replace the three plot type functions used for a high-resolution screen.

The following program will demonstrate the colorful use of MULTI by drawing a rectangle in red on the left side of the screen, another rec-

tangle in blue on the right side of the screen, and a purple line between them. **Clear computer memory**, for automatic line numbering use AUTO 1Ø,1Ø, and enter the following program. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Command Illustrated	Program Listing
HIRES and MULTI	1Ø HIRES Ø,13:MULTI 2,6,4
REC	2Ø REC 1Ø,8Ø,4Ø,3Ø,1
LINE	3Ø LINE 6Ø,95,1ØØ,95,3
REC	4Ø REC 11Ø,8Ø,4Ø,3Ø,2
	5Ø GOTO 5Ø

The first rectangle plotted (line 2Ø) will use plot type 1, so it will be drawn using the first MULTI parameter, color 2—red. The LINE statement (line 3Ø) will use plot type 3, so the line will be drawn using the third MULTI parameter, color 4—purple. The second rectangle plotted (line 4Ø) uses plot type 2, thus specifying the use of the second MULTI parameter, color 2—blue. Both REC and LINE function the same on either graphic screen—high-resolution or multicolor. The plotting locations for these three graphic shapes are illustrated on the multicolor screen grid shown below, Figure 18.3.

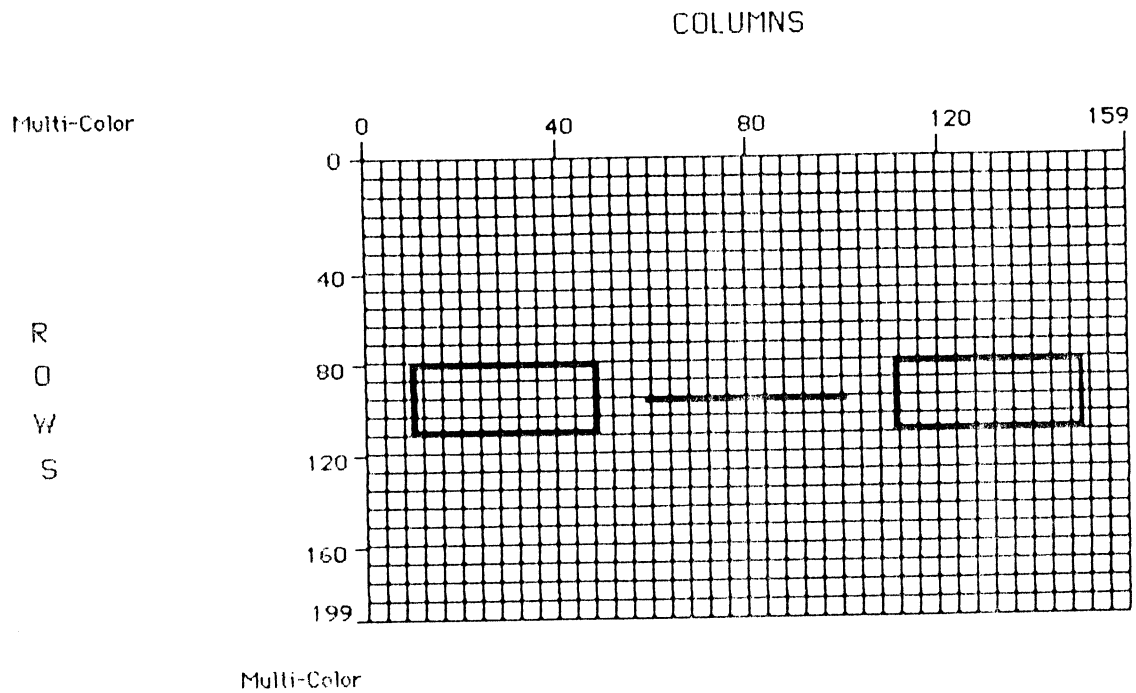


Figure 18.3. Two rectangles and a line plotted on a grid using the multicolor screen configuration.

RUN the program. As you can see, the screen background color (light green) specified in the HIRES statement on line 1Ø was ignored. The background color used was the normal Simons' BASIC background color of light grey. The plotting color given in HIRES (black) was also ignored. Press the RUN/STOP key to end the program and return to the low-resolution screen. To add screen color to this program,

ENTER:
```
5 COLOUR 5,13
```

and RUN the program again. Now the multicolor graphics appear on a light green screen with a green border. Press the RUN/STOP key to end the program.

LIST the program. To use the LINE statement to demonstrate all five MULTI plot types, edit line 3Ø to change plot type from 3 to the variable PT and add lines 5Ø through 7Ø, as shown below.

```
3Ø LINE 6Ø,95,1ØØ,95,PT
5Ø PT=PT+1:PAUSE 3
6Ø IF PT<5 GOTO 2Ø
7Ø END
```

The plot type of LINE will change five times, with a pause of three seconds between the display of each plot type. Plot type will begin at Ø—clears a dot—showing no line on the graphic screen, then use plot type 1 (color 1, red), plot type 2 (color 2, blue), plot type 3 (color 3, purple), and end with plot type 4 which reverses plot type 3 to plot type Ø.

Squares on a Multicolor Screen

In order to use the REC command to plot a square on the multicolor screen configuration, the ratio of physical screen (TV or monitor) width to depth is applied to the depth, rather than the width. This is necessary because, unlike the high-resolution screen, the multicolor screen configuration is deeper than it is wide—200 rows by 160 columns. The same adjustment factor used for the high-resolution screen—1.4—will be used for the multicolor screen, but the factor is applied to the depth rather than the width parameter.

LIST the program in memory. By changing the depth parameter in the two REC statements as shown below, the two rectangles will be replaced by squares. Rather than again displaying the five plot types in the LINE statement, the program will pause for five seconds after printing the squares, then end. The Simons' BASIC commands illustrated are shown to the left of the program statement in which they occur. Enter the new statements.

Command Illustrated	Program Statements
REC (plot a square)	20 REC 10,80,40,40*1.4,1
REC (plot a square)	30 REC 110,80,40,40*1.4,2
	40 PAUSE 5:END

Circles, Angles, and Arcs

To plot a circle on a high-resolution screen, the horizontal radii had to be adjusted—multiplied by 1.4. For a multicolor screen, the vertical radii must be adjusted by the same amount, that is, vertical radii=horizontal radii*1.4. If this adjustment isn't best for your monitor, experiment in the range 1.4 to 1.6 and choose the one that works best for you. (**Note**: When producing a hard copy of a graphic screen, the radii must be equal in order to print a perfect circle.)

The circle will very clearly show the difference in extent of resolution between the high-resolution and multicolor screen. Add the following statement lines to the program in memory. Line 40 will replace the PAUSE and END statements.

```
40 CIRCLE 80,100,25,25*1.4,3
50 GOTO 50
```

RUN the program again. You are now seeing the "jaggies", the staircase effect of lower resolution. This is a very different circle than plotted in high-resolution! Press the RUN/STOP key to end the program.

This would not make the best clock face, as you will see by adding the following statements to plot the hour and minute hands.

```
50 ANGL 80,100,0,20,20*1.4,2:ANGL 80,100,90 20,20*
1.4,2
60 GOTO 60
```

Of course, the same adjustment is made to the vertical radii in the ANGL command as was made in the CIRCLE command.

RUN the program and press the RUN/STOP key to end.

For a demonstration of the ARC command on a multicolor screen, add the following statement lines to the program in memory.

```
60 ARC 80,160,0,180,1,15,15*1.4,2:PAUSE 3
70 ARC 80,160,180,360,1,15,15*1.4,3:PAUSE 3
```

Two arcs will be plotted to form a smaller circle under the center circle. After each arc (half-circle) is plotted, the program will pause for three seconds. RUN the program.

Changing Colors

As you know, the LOW COL command has three color parameters, but is only able to use the first two in high-resolution mode. Now you will be able to make use of all three parameters.

To change the colors used in the program in memory add the following statement lines.

```
80 LOW COL 7,0,3
90 X=X+1:IF X=1 GOTO 30
100 PAUSE 3:END
```

The shapes will be plotted a second time, using the colors yellow, black, and cyan, the program will pause three seconds, and then it will end. To speed-up the changing colors, you may want to remove the PAUSE 3 statements following the ARC statements in lines 60 and 70.

RUN the program.

Painting Shapes

You already know how to use the PAINT command, but now it can be used to both paint and **repaint** shapes on the multicolor screen.

ENTER:
```
100 PAINT 15,85,2:PAINT 115,85,3:PAINT
90,90,1:PAINT 90,150,1:PAUSE 3
110 PAINT 15,85,1:PAINT 115,85,2:PAINT
90,90,3:PAINT 90,150,3:PAUSE 5
```

LIST the program. After the program plots the four shapes a second time using LOW COL, line 100 will paint the left square black, the right square blue, and the two circles yellow. Then execution will pause for three seconds. Line 110 will repaint the left square yellow, the right square black, and the two circles blue. Program execution will pause for five seconds, and then end. Note that the color yellow appears orange when painted on a light green background.

RUN the program.

Remember that all of the plotting commands are used for both graphic screens—the high-resolution screen and the multicolor screen! Only the additional multicolor graphic commands and those requiring changes for use on a multicolor screen were reviewed in this section.

VII Sprite Graphics

Introduction

This section is written to be easily understood by the Commodore 64 BASIC programmer who has not used the PEEKs and POKEs of Commodore BASIC's sprite graphics. Those of you who have been creating sprites with BASIC will anticipate the commands that are needed and know which POKEs and PEEKs are being replaced by the Simons' BASIC commands. You are also familiar with the screen configuration and already know how to position sprites on the screen. Screen configuration and sprite positioning are reviewed in detail for the sprite novice. For both the newcomer and the knowledgeable sprite user, Simons' BASIC will eliminate much of the tedious sprite programming. But—in Simons' BASIC you will have to get used to calling a sprite a MOB. However, MOB—standing for Movable Block Object—is certainly not as exciting a word as sprite! The words MOB and sprite are used interchangeably throughout this section.

The following is a brief overview of the sprite commands reviewed in this section.

Chapter 19—Creating and Using Sprites

DESIGN specifies sprite resolution—high resolution or multicolor—and reserves 64 bytes of memory space for the sprite design.

@ is used to begin each statement of the design grid for both high-resolution and multicolor sprites.

MOB SET initializes a high-resolution or multicolor MOB.

CMOB is used to specify the two additional colors to be used in a multicolor sprite.

MMOB displays a high resolution or multicolor MOB. This command also allows the user to specify an expanded sprite size, and move the sprite on the screen at a user-defined speed.

RLOCMOB is used to move a displayed MOB to another screen location.

DETECT is used for collision detection between two sprites, or between any displayed sprite and screen data.

CHECK is the collision testing statement.

MOB OFF clears a MOB from the screen.

Chapter 20—Sprites on a High-Resolution or Multicolor Screen

This chapter reviews the MOB data storage address changes in the DESIGN and MOB SET command parameters necessary for displaying sprites on a graphic screen—high resolution or multicolor. Also reviewed is the required statement order for displaying high-resolution and/or multicolor sprites on each of the three screen types.

Key to Entering Program Listings

In this section, special instructions are given to the right of design statements for entering design grid data. The code to be entered is followed by the number of repetitions of that code. Each code and its repetitions is separated from the next by a comma. If two codes are to be repeated, they are shown together, followed by the number of repetitions.

For example: **.20,B,.3** means to enter twenty periods, one B, and three periods.

19 Creating and Using Sprites

This chapter reviews each statement used for the design, display, and interaction of "sprites"—those lovable creatures so well known from arcade games. A sprite is a programmable character that can assume any user-defined shape. A sprite is known as a MOB—Movable Object Block—in Simons' BASIC. You have been introduced to sprites in low resolution, using the DESIGN 2 and @ commands for creating user-defined characters. A sprite design, however, is much larger in size, and, therefore, in detail, than a user-defined character. Sprites (MOBs) can be either high-resolution or multicolor, be stationary or move rapidly, can be expanded in size, and can collide and disappear. A maximum of eight sprites may be displayed at any one time.

MOBs can be displayed on normal and graphic screens—low resolution, high-resolution, and multicolor. This chapter will review each MOB statement and display the MOBs on a low resolution screen. Chapter 20 will review the use of sprites on high-resolution and multicolor screens.

The DESIGN statement has the same function in sprite design as the DESIGN 2 statement has in user-defined graphics, and the @ statement also has the same function for both design types. Only the parameters change.

DESIGN

This command reserves 64 bytes of memory in RAM (Random Access Memory) for each MOB created. DESIGN must always **directly precede the @ statements** in which the design is created. If the DESIGN statement follows the @ statements, a SYNTAX ERROR message will be given for the line number of the first @ statement. If other

statements appear between the DESIGN statement and the first @ statement, the SYNTAX ERROR will be given for the DESIGN statement.

Statement format:
DESIGN resolution,starting memory location

The resolution parameter specifies either a high resolution or multicolor MOB, **not screen type**, with the number 0 indicating high resolution and the number 1 specifying multicolor. The @ command, reviewed next, is used to design these two MOB types. The difference between the MOB types is the same as the difference between high-resolution and multicolor plotting graphics. multicolor sacrifices resolution for additional color. In sprites, however, the resolution difference is not as obvious.

Starting memory location is the address in memory where the MOB data will be stored. MOBs using the low-resolution screen use a portion of memory normally used for BASIC programs. For this reason, the portion of memory allocated to MOBs (sprites) must be below the BASIC program creating them, in order for the program to be stored in memory. A BASIC program is stored in memory beginning at memory location 2048. MOB memory locations begin at 8192. If the first memory location available for a MOB is used, there will be 6,144 bytes (8192-2048) available for storing the program. If additional memory is needed, specify a higher memory address for the first MOB to allow sufficient memory for your program. The last address available for a MOB is 16320, leaving 14,217 bytes available for a program using one MOB. Because of the MOB memory locations used by Simons' BASIC, this is a significant change in available BASIC memory.

Note: If there isn't sufficient memory to store your program, the program will be lost after it is run. For this reason, it's a good idea to SAVE your program before running it, unless you're sure that your program is sufficiently short and will not overlap the memory space reserved for that program's MOBs, as specified in the program's DESIGN statement(s). The over 14,000 bytes should be sufficient for most of your programming needs. However, if your program, including the reserved space for MOBs, is too large to be stored in this area of memory, you will have to either move your BASIC program to another portion of memory **or** use a graphic screen. When using a high-resolution or multicolor screen, this problem is eliminated, as described in Chapter 20, pages 207-208.

The block number of the starting memory location specified in this command is a parameter of the command that initializes the MOB. This initializer—MOB SET—is reviewed later in this chapter. The memory locations and block numbers available for MOBs used on a low-resolution screen are:

Block Numbers	Memory Locations
128-255	8192-16320

Block number * 64 = memory location, for example, block number 128 *64=memory location 8192, block number 129*64=memory location 8256, and so on. Figure 19.7 at the end of this chapter, pages 204-206, shows each available starting memory location and block number.

A program may have as many MOBs as there is available memory space. This means a whopping maximum of 128 MOBs! However, no more than eight MOBs can be displayed at any one time.

@

This command serves the same function for sprite graphics that it does for low resolution user-defined characters. However, you will be using a much larger grid to design sprites. For a high-resolution sprite, the grid is 24 pixels (dots) wide by 21 lines deep. A multicolor sprite is half the width and the same depth—12 pixels wide by 21 lines deep. Both the high-resolution and multicolor sprite will occupy the same space on the screen, since each multicolor horizontal dot—represented by one dot on the design grid—is two pixels wide.

Statement format:
@ design

For a high-resolution sprite, the design is drawn by using the same codes used for low resolution user-defined characters. That is, a period for turned off pixels and the letter B for turned on pixels. Only one color is allowed in high-resolution sprites.

For a multicolor sprite, three colors may be used for the pixels that are on. The letter C designates color 1, the letter D designates color 2, and the letter B is used to designate the main color, color 3.

It is important to keep all 21 lines of a MOB positioned the same by

using line numbers containing the same number of digits. First, it is easier to design the MOB by having the grid line up exactly on the screen. Secondly, if you use line numbers containing a different number of digits (ex., 8Ø, 9Ø, 1ØØ, 11Ø, and so on), and it is necessary to edit one or more @ statements after running the program, you may have a problem. When you LIST the program, the @ sign on the higher digit lines will have disappeared! You may have to reinsert the @ signs in order to have the program run properly. The disappearance of the @ sign will not regularly "crash" the program, but it sometimes will. Better safe than sorry!

If a SYNTAX ERROR message is given for the first @ statement, it can be caused by either the improper location of the DESIGN statement in your program or by an incorrect number of pixels specified in the @ statement.

Note: If a comma is used rather than a period in an @ statement, the program will "hang" when run. If this occurs, press the RUN/STOP and RESTORE keys, LIST the program, and replace the comma with a period.

The following statements will begin the demonstration program, creating a high-resolution MOB beginning at memory location 8192 and a multicolor MOB beginning at memory location 8256. The high-resolution MOB is a cluster of three stars. The multicolor MOB is a spiral galaxy, which will be displayed in three colors. The sprites will be displayed on a dark gray screen having a black border. The multicolor MOB will not look like a spiral in the design grid. Since the 12 by 21 dot grid is rather limiting, a later command will be used to enlarge this sprite horizontally.

Clear computer memory. For automatic line numbering with the following program, use AUTO 1ØØ,1Ø. In order to simplify entering the @ statements, a guide to entering these statements is shown to the right of the program statements. This keying in procedure was described in the section introduction, under Key to Entering Program Listings, page 188. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Commands Illustrated	Program Statements	Entry Guide
	1ØØ PRINT "[CLEAR]"	
COLOUR	11Ø COLOUR Ø,11	
DESIGN	12Ø DESIGN Ø,8192	
@	13Ø @.....B...	.2Ø,B,.3

```
@          140 @.................B.B.B.  .18,B.3
@          150 @.................BBB..  .19,B3,.2
@          160 @.................BBBBBBB .17,B7
@          170 @.................BBB..  .19,B3,.2
@          180 @.................B.B.B.  .18,B.3
@          190 @.................B...    .20,B,.3
@          200 @...B.....................  .3,B,.20
@          210 @.B.B.B.................... .B3,.18
@          220 @..BBB.....................  .2,B3,.18
@          230 @BBBBBBB..................  B7,.17
@          240 @..BBB.....................  .2,B3,.19
@          250 @.B.B.B.................... .B3,.18
@          260 @...B.....................  .3,B,.20
@          270 @............B.........    .13,B,.10
@          280 @..........B.B.B........   .11,B.3,.7
@          290 @...........BBB.........   .12,B3,.9
@          300 @.........BBBBBBB.......   .10,B7,.7
@          310 @...........BBB.........   .12,B3,.9
@          320 @..........B.B.B........   .11,B.3,.7
@          330 @............B.........    .13,B,.10
DESIGN     340 DESIGN 1,8256
@          350 @.....CCBBB..             .5,C2,B3,.2
@          360 @...CC.......             .3,C2,.7
@          370 @..CC........             .2,C2,.8
@          380 @.CC.........             .,C2,.9
@          390 @CC..........             C2,.10
@          400 @.CC..BBBBBB.             .,C2,.2,B6,.
@          410 @.C..C.......             .,C,.2,C,.7
@          420 @C..C....BBB.             C,.2,C,.4,B3,.
@          430 @.C.B...BB...             .,C,.,B,.3,B2,.3
@          440 @C..DDDDD....             C,.2,D5,.4
@          450 @.BDDDCCDDCC.             .,B,D3,C2,D2,C2,.
@          460 @.BDDCCCDD.CC             .,B,D2,C3,D2,.,C2
@          470 @..BDDDDD.CC.             .2,B,D5,.,C2,.
@          480 @B..DDD.BB.CC             B,.2,D3,.,B2,.,C2
@          490 @BB....BB.CC.             B2,.4,B2,.,C2,.
@          500 @.BB.BBB..CC.             .,B2,.,B3,.2,C2,.
@          510 @.BBBBB..CC..             .,B5,.2,C2,.2
@          520 @.......CC...             .7,C2,.3
@          530 @......CC....             .6,C2,.4
@          540 @.....CC.....             .5,C2,.5
@          550 @...BBB......             .3,B3,.6
```

These statements will not display the sprites. The above statements designed two MOBs and allocated memory space for them. The next step is to initialize each MOB.

MOB SET

This command initializes both high-resolution and multicolor MOBs.

Statement format:
MOB SET number,block,color,priority,resolution

The numbers that you choose for each MOB will determine their screen priority over one another. The MOB having the lower number has priority. If two or more MOBs are moving and their screen paths cross, the lower numbered MOB will pass in front of the higher numbered MOB. **Only the numbers Ø through 7 may be used.** Remember, the maximum number of MOBs that may be displayed at one time is eight. If a number higher than 7 is used, the program either will not run properly or will not run at all! In either case, if you try to run the program again or edit the program, the system will "hang." The only remedy is to turn off the computer and begin again, hopefully by reloading the program and correcting the MOB number before running the program.

The block parameter designates the memory block from which the MOB design data will be taken. This is the block number of the starting memory location specified in the MOB's DESIGN statement. If a block number greater than 255 is used, an ILLEGAL QUANTITY ERROR will be given. However, **if an incorrect block number is used, no error message is given but the MOB will be distorted**. The block numbers and their memory locations are shown in Figure 19.7, pages 204-206, at the end of this chapter.

The color parameter specifies the color for a high-resolution MOB—the B dots of the design grid. For a multicolor sprite, the color parameter designates color 3—the C dots of the design grid. (Colors 1 and 2 of a multicolor sprite are designated in the CMOB command, which will be reviewed next.) The Commodore colors are shown in Figure 13.1, page 108.

The priority parameter designates whether screen data has priority over MOBs. If MOBs have priority, they pass OVER screen data. If

screen data has priority, MOBs pass UNDER the screen data. This, of course, means that the MOB will be partially or fully hidden when passing. The number Ø specifies MOB priority, and the number 1 specifies screen data priority.

The resolution command specifies whether the MOB is high resolution or multicolor. The number Ø indicates high resolution, and the number 1 indicates multicolor. This is the same parameter used in the DESIGN command.

The following two statements will initialize the two MOBs in the demonstration program. Line 56Ø initializes the high-resolution MOB—a cluster of three white stars. This MOB will have priority over the multicolor MOB—a spiral galaxy, but neither MOB will have priority over screen data. The multicolor MOB is initialized in line 57Ø.

Enter the following statements. The Simons' BASIC commands illustrated are shown to the left of the program statements.

Command Illustrated	Program Statements
MOB SET	56Ø MOB SET Ø,128,1,1,Ø
MOB SET	57Ø MOB SET 1,129,7,1,1

The MOB color for high-resolution MOBs is specified only in the MOB SET statement. Only color C of the multicolor MOB—in this program the core and portions of the spirals—is specified in MOB SET. Color C will use yellow. The two additional colors available to multicolor MOBs must be set in a second statement.

CMOB

This statement specifies the two additional colors available to multicolor MOBs.

Statement format:
CMOB color B, color D

If this statement is not used—even though colors B and D were used in the design grid—no error message will be given. But the pixels coded B will not be displayed in the desired color, and the pixels coded D will be off. (In this program, purple was used for color B and the pixels coded D were off when the CMOB statement was not used.)

Now we are ready to display the MOBs. Add the following CMOB statement to the program in memory to color the rest of the spiral.

```
ENTER:    580 CMOB 15,10
```

The pixels coded B—portions of the spiral—will be displayed in light gray, and the pixels coded D—those surrounding the core—will be displayed in light red.

The sprites still cannot be displayed without one more statement. If the program is run now, only a dark gray screen (specified in the COLOUR statement) will be displayed.

MMOB

This is the command used to display a MOB. The MOB may be stationary or moved on the screen at a user-specified speed. A MOB may also be expanded in size—either in width, height, or both.

Statement format:
MMOB number,x,y,x1,y1,expansion,speed

The MOB number is the same number that was used in the MOB SET command to initialize the sprite.

As always, the parameters x and y specify the starting column and row coordinates, with x specifying column and y specifying row. This is the location that will be used for the first display of the MOB. The **top left** corner of the MOB will begin at this starting location. The top left corner is the first pixel—whether that pixel is on (B, C, or D) or off (a period)—in the design grid. The parameters x1 and y1 specify the column and row respectively to which the MOB will be moved. Again, the **top left** corner of the MOB will appear at this location. To specify a stationary MOB, simply use the same locations for both x,y and x1,y1.

The bit-mapped screen used by sprites is identical in pixel size to the high-resolution screen that you have already used, that is, 320 columns by 200 rows. However, the beginning and ending column and row numbers change. The columns are numbered 24 through 343, and the rows 50 through 249. The first off-screen positions are 344 horizontally, and 250 vertically. As an aide to correctly positioning MOBs, this screen configuration is shown below in Figure 19.1.

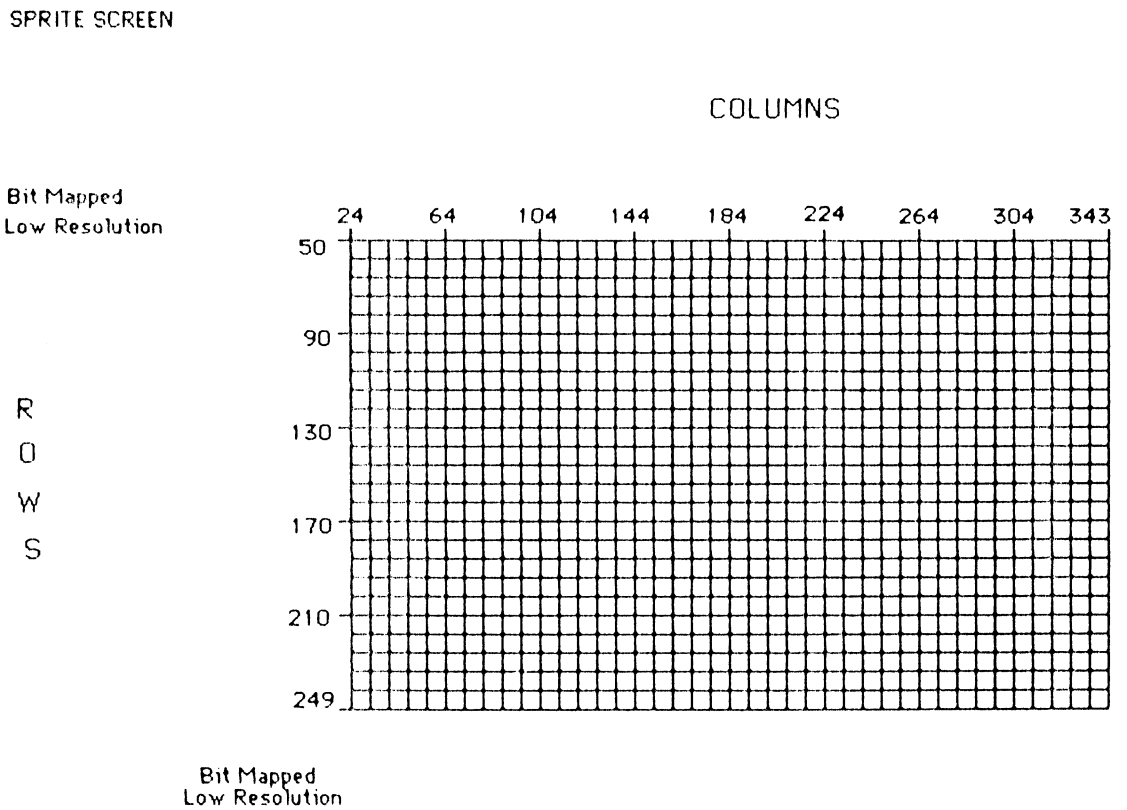


Figure 19.1. Sprite Screen Grid—columns and rows numbered.

The expansion parameter specifies the desired display size of the MOB. The expansion numbers used for this parameter, and the resulting display size, are shown below.

Expansion Number	Display Size
Ø	**Normal size**—as created on the design grid—24 dots by 21 dots.
1	**Twice the width** displayed on the design grid, but the same height—48 dots by 21 dots.
2	Normal width, but **twice the height** displayed on the design grid—24 dots by 42 dots.
3	**Twice the width and height** displayed on the design grid—48 dots by 42 dots.

Figure 19.2.

In positioning an expanded MOB, this size difference must be part of the calculation for specifying the MOB locations. For your convenience, the corner locations for a MOB to be shown fully on the screen are illustrated below on screen grids—one for each expansion parameter—in Figures 19.3-19.6.

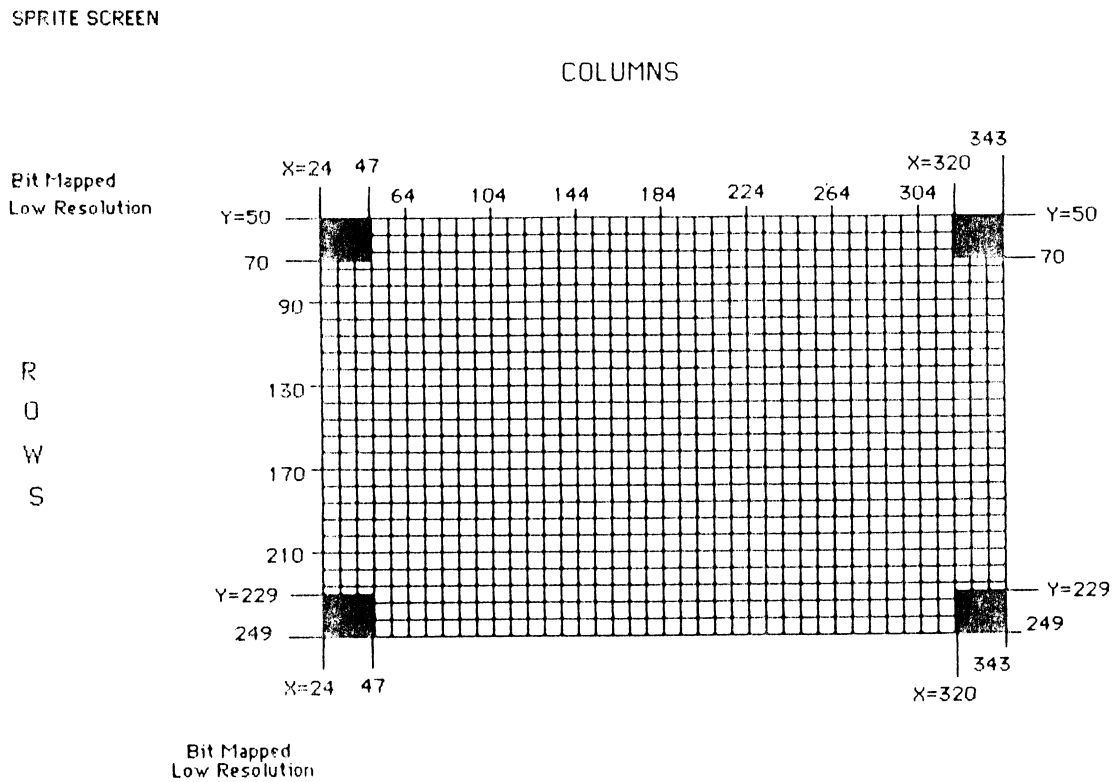


Figure 19.3. Corner screen locations for MOB expansion size Ø.

As shown on the following grids, a screen location adjustment of 21 pixels is required for a MOB expanded in height, and 24 pixels for a MOB expanded in width. Because MOB location is specified using the top left corner of a MOB, corner positioning for expanded height requires adjustment of the y1 (ending row) parameter, while expanded width requires adjustment of the parameters x or x1 (column locations) for the left edge of the screen.

The location used to move any size MOB fully off the screen at the top, bottom, and right side is the same. However, only a MOB of **normal width**—expansion sizes Ø and 2—can be moved fully off the left side of the screen. The column or row parameters used are shown on page 200, following Figure 19.6.

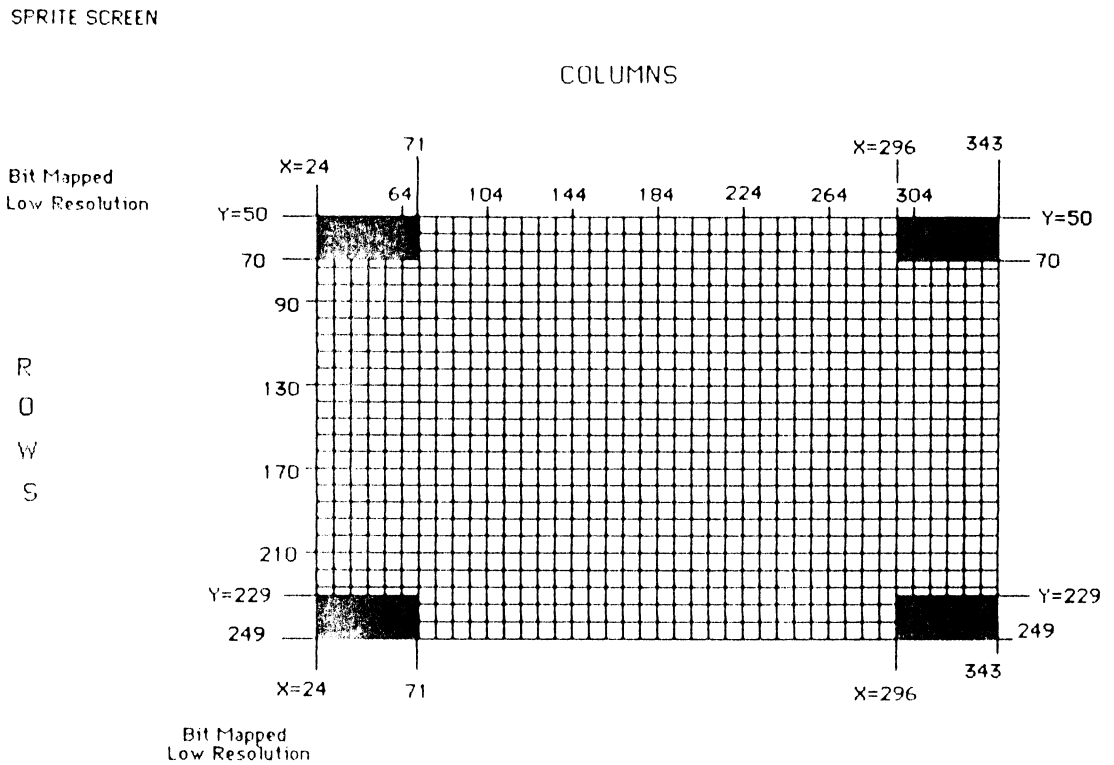


Figure 19.4. Corner screen locations for MOB expansion size 1.

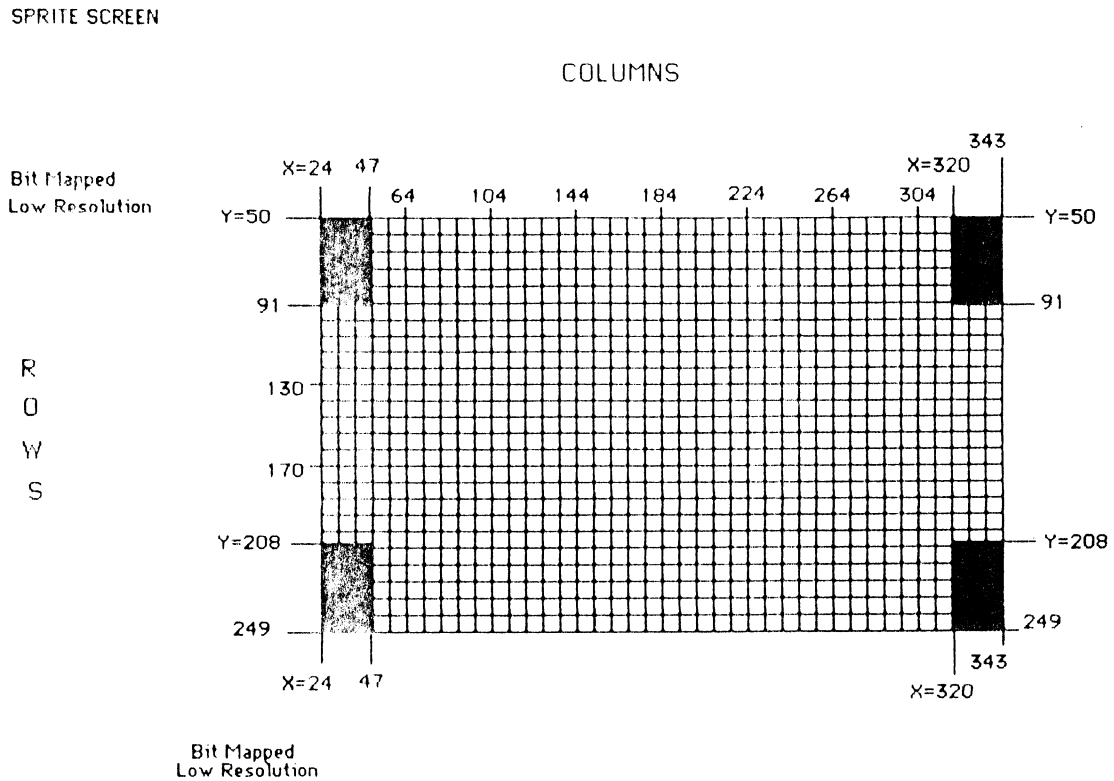


Figure 19.5. Corner screen locations for MOB expansion size 2.

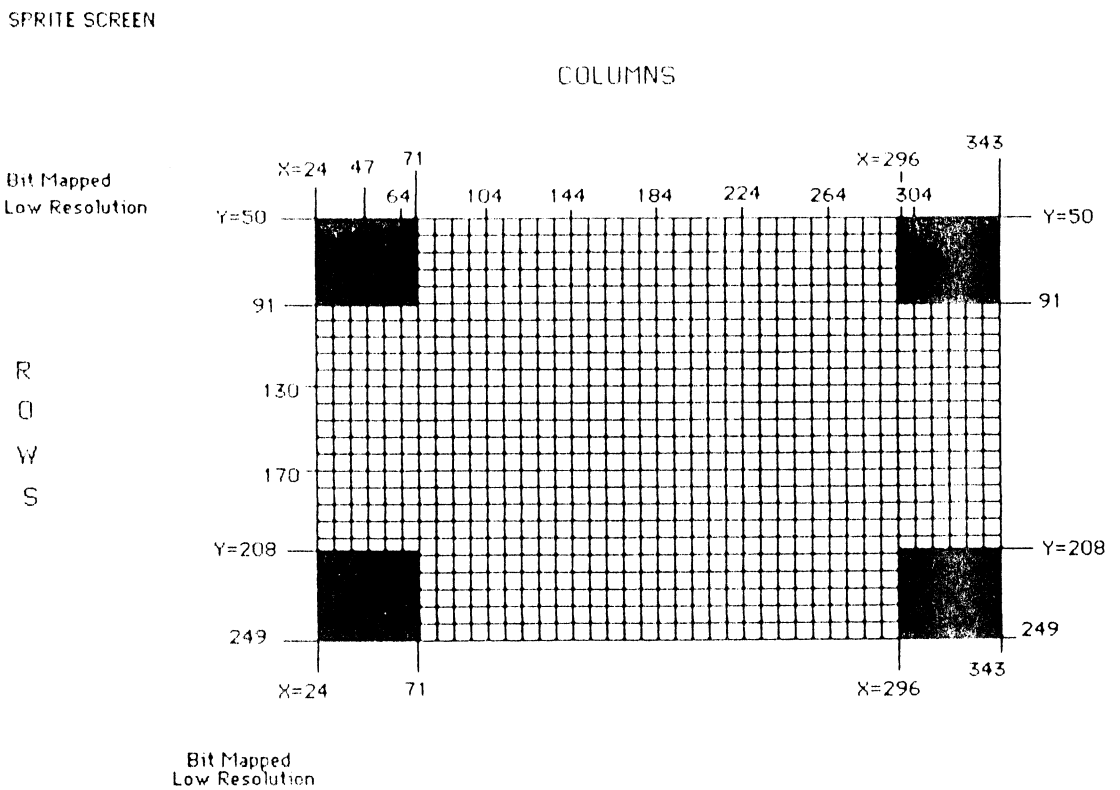


Figure 19.6. Corner screen locations for MOB expansion size 3.

- Ø for the column parameter to move a **normal width** MOB off the left side of the screen. MOBs expanded in width—expansion 1 and 3—cannot be moved fully off the left side of the screen.
- 344 for the column parameter to move off the right side of the screen.
- Ø for the row parameter to move off the top of the screen.
- 25Ø for the row parameter to move off the bottom of the screen.

The speed parameter specifies the speed at which the MOB will move from x,y to x1,y1. Speed is designated by the numbers 1 through 255, with 1 used for the fastest speed and 255 used for the slowest speed. The speed parameter must always be used, even though the sprite is to remain stationary. If no movement is required, the number used does not matter as long as it is in the range 1-255.

Let's finally display and move sprites! As you will see, sprite movement is very smooth. Add the following MMOB statements to the program in memory.

ENTER:

```
59Ø MMOB 1,17Ø,15Ø,17Ø,15Ø,1,255
6ØØ MMOB Ø,Ø,Ø,24Ø,2ØØ,Ø,2ØØ
61Ø MMOB Ø,28Ø,2ØØ,Ø,Ø,Ø,1ØØ
```

MOB 1—the spiral galaxy—will be displayed at twice the normal width and remain stationary in the center of the screen (line 59Ø), while MOB Ø—the star cluster—will provide the movement. Line 6ØØ will send the star cluster diagonally across the screen from the upper left-hand corner, passing in front of the spiral galaxy, to the bottom right side of the screen. The MOB will be displayed at normal size and travel at a speed of 200. Line 61Ø will send the star cluster back along the path traveled in the previous statement, at twice the speed of the first movement, and off the screen.

RUN the program. Press the RUN/STOP and RESTORE keys to return to the normal screen.

The next command is used to relocate a displayed MOB.

RLOCMOB

This command enables you to move an on-screen MOB to new screen location. The current MOB location is not required.

Statement format:
RLOCMOB number, x1,y1,expansion,speed

All of these parameters are the same as those used in the MMOB command. As in the MMOB command, the parameters x1 and y1 are the column and row coordinates for the location to which the MOB will be moved.

Add the following statement to the program in memory to move the spiral galaxy from its current location to the bottom left-hand corner of the screen.

ENTER:
```
62Ø RLOCMOB 1,24,229,1,8Ø
```

RUN the program again and press the RUN/STOP and RESTORE keys to return to the normal screen.

On to collisions!

DETECT

If collision detection is used, this command must be used twice. Its first use will clear the sprite control register, so that the second use can

initialize collision detection. DETECT handles two types of collisions: sprite to sprite collision, and sprite and screen data collision.

Statement format:
For sprite to sprite collisions: DETECT Ø
For sprite and screen data collisions: DETECT 1

Add the first DETECT statement to clear the sprite control register, as shown below.

ENTER: `630 DETECT Ø`

On to collision testing.

CHECK

This command is used for collision testing either between two MOBs or between a MOB and screen data.

Command format
Collision test for two sprites:
IF CHECK(MOB number, MOB number) = Ø THEN statement(s)

The MOB numbers are the only parameters used by CHECK for sprite collision detection. The MOB numbers specified here are the same numbers that were used to identify each of the sprites when they were initialized. Only a collision between the two sprites specified will be detected. A collision is detected when an "on" pixel of one of the sprites specified overlaps an "on" pixel of the other sprite specified.

Collision test for sprite and screen data:
IF CHECK(Ø) = Ø THEN statement(s)

The zero parameter of CHECK initiates collision detection between any sprite and screen data. A collision is detected when an "on" pixel of any sprite overlaps a character on the screen.

MOB OFF

This command clears the specified MOB from the screen.

Statement format:
MOB OFF number

The parameter number specifies the number of the MOB to be removed from the screen.

Add the following statements to the program in memory to cause and detect a collision between the two sprites, and remove sprite Ø—the star cluster—from the screen. You may use AUTO 64Ø,1Ø for automatic line numbering. The Simons' BASIC commands illustrated are shown to the left of the program statement in which they occur.

Command Illustrated	Program Statements
	64Ø FOR I=1 TO 1Ø
	65Ø X=X+1Ø
DETECT and CHECK	66Ø DETECT Ø:IF CHECK(Ø,1)=Ø THEN 7ØØ
RLOCMOB	67Ø RLOCMOB Ø,24,5Ø+X,Ø,1Ø
RLOCMOB	68Ø RLOCMOB 1,24,229-X,1,1Ø
	69Ø NEXT I
MOB OFF	7ØØ MOB OFF Ø
PRINT AT	71Ø PRINT AT(1Ø,1Ø)"GALAXY RULER"
RLOCMOB	72Ø RLOCMOB 1,2ØØ,12Ø,1,8Ø

Text has been added to the screen display using the PRINT AT statement—a low resolution statement. The high-resolution TEXT statement is not used in this program because the program is being run on a low-resolution screen. The spiral galaxy—MOB 1—will pass behind this screen data, as specified in the priority parameter of the MOB SET statement.

RUN the program and press the RUN/STOP and RESTORE key to return to the normal screen.

Any number of low-resolution screen statements—in addition to the PRINT AT statement—may be added to this program. This includes all of the statements reviewed in Sections I through IV of this book. For

example, the user-defined characters reviewed in Chapter 14 can be used to produce interesting effects when combined with sprites on a low-resolution screen.

If you are not going on to the next chapter immediately, **SAVE** this program. It will be used again in the next chapter to demonstrate the use of sprites with plotting graphics on a high-resolution screen.

Block Numbers	Starting Memory Locations	Block Numbers	Starting Memory Locations
128	8192	155	992Ø
129	8256	156	9984
13Ø	832Ø	157	1ØØ48
131	8384	158	1Ø112
132	8448	159	1Ø176
133	8512	16Ø	1Ø24Ø
134	8576	161	1Ø3Ø4
135	864Ø	162	1Ø368
136	87Ø4	163	1Ø432
137	8768	164	1Ø496
138	8832	165	1Ø56Ø
139	8896	166	1Ø624
14Ø	896Ø	167	1Ø688
141	9Ø24	168	1Ø752
142	9Ø88	169	1Ø816
143	9152	17Ø	1Ø88Ø
144	9216	171	1Ø944
145	928Ø	172	11Ø/8
146	9344	173	11Ø72
147	94Ø8	174	11136
148	9472	175	112ØØ
149	9536	176	11264
15Ø	96ØØ	177	11328
151	9664	178	11392
152	9728	179	11456
153	9792	18Ø	1152Ø
154	9856	181	11584

Figure 19.7. Table of Block Numbers and Starting Memory Locations for MOBS

Block Numbers	Starting Memory Locations	Block Numbers	Starting Memory Locations
182	11648	216	13824
183	11712	217	13888
184	11776	218	13952
185	1184Ø	219	14Ø16
186	119Ø4	22Ø	14Ø8Ø
187	11968	221	14144
188	12Ø32	222	142Ø8
189	12Ø96	223	13272
19Ø	1216Ø	224	14336
191	12224	225	144ØØ
192	12288	226	14464
193	12352	227	14528
194	12416	228	14592
195	1248Ø	229	14656
196	12544	23Ø	1472Ø
197	126Ø8	231	14784
198	12672	232	14848
199	12736	233	14912
2ØØ	128ØØ	234	14976
2Ø1	12864	235	15Ø4Ø
2Ø2	12928	236	151Ø4
2Ø3	12992	237	15168
2Ø4	13Ø56	238	15232
2Ø5	1312Ø	239	15296
2Ø6	13184	24Ø	1536Ø
2Ø7	13248	241	15424
2Ø8	13312	242	15488
2Ø9	13376	243	15552
21Ø	1344Ø	244	15616
211	135Ø4	245	1568Ø
212	13568	246	15744
213	13632	247	158Ø8
214	13696	248	15872
215	1376Ø	249	15936

Figure 19.7. Table of Block Numbers and Starting Memory Locations for MOBS—cont.

Block Numbers	Starting Memory Locations	Block Numbers	Starting Memory Locations
25Ø	16ØØØ	253	16192
251	16Ø64	254	16256
252	16128	255	1632Ø

Figure 19.7. Table of Block Numbers and Starting Memory Locations for MOBS

20 Sprites on a High-Resolution or Multicolor Screen

Sprites may also be used on high-resolution and multicolor screens, giving you the ability to include sprites with plotting graphics. For both of these screen configurations, MOB design data is stored in a different portion of memory than is used for sprites on a low-resolution screen. This changes both the memory address used in the DESIGN statement and the associated block number used in the MOB SET statement. When using **both** high-resolution and multicolor MOBs on either of these screen types, a specific placement of the MOB statements within the program is required.

DESIGN and MOB SET Statements

In order to access a graphic screen, the MOB design data is stored in a different portion of RAM. This is done by adding a graphic-constant value to a low address in the normal BASIC program space.

The block numbers and associated memory locations used on a high-resolution or multicolor screen are:

Block Numbers	Memory Locations
32-63	2048-4032

The graphic constant—$C000 in hexadecimal or 49152 in decimal—added to these memory locations moves the MOB data out of the way of the BASIC program space. The DESIGN statement used with high-resolution and multicolor screens is shown below.

Statement format:
DESIGN resolution,starting memory location + graphic constant value

Because of the addition of a graphic constant value of decimal **49152 or $C000** in hexadecimal, the lowest memory location used for MOB data when using a graphic screen will be 51200. This then eliminates the problem of memory limitations on your BASIC program, explained on pages 190-191. The memory that was unavailable to your program when using a low resolution screen is now available again.

The MOB SET parameters do not change. Only the block number used changes. The graphic constant value is **not** added to the block number.

Statement Order

When displaying **only** high-resolution MOBs on a high-resolution screen, the order of the graphic statements and the MOB statements is unimportant. That is, the graphic statements may precede and/or follow the MOB statements. The same is true when using **only** multicolor sprites and graphics on a multicolor screen.

However, to display **both** high-resolution and multicolor sprites on either a high-resolution or multicolor screen, the graphic statements—whether high-resolution or multicolor—**must precede the first DESIGN statement**. If they follow the first DESIGN statement, *no error message will be given, but the MOBs will not be recognizable.*

The use of high-resolution graphic statements and both a high-resolution and a multicolor sprite will be demonstrated by editing the program from the previous chapter. First, the DESIGN and MOB SET statements must be changed for use on a high-resolution screen. EDIT or ENTER the following statement lines.

```
120 DESIGN 0,32*64+49152
340 DESIGN 1,33*64+$C000
560 MOB SET 0,32,1,1,0
570 MOB SET 1,33,7,0,1
```

The DESIGN statements (120 and 340) calculate the decimal starting location by multiplying the block number by the number of bytes per block. The first DESIGN statement uses the graphic constant in decimal form, and the second uses the graphic constant in hexadecimal form. Note that the priority of MOB 1 over screen data was changed in its MOB SET statement (the fourth parameter, line 570) giving the spiral galaxy priority over screen data. This sprite will now pass over all screen data, including the high-resolution graphics.

The following statements will add high-resolution plotting to create a quarter moon and sun as additional screen graphics before initializing or displaying the MOBs.

ENTER:

```
112 HIRES 7,11
114 ARC 80,140,110,0,1,16*1.4,16,1
116 ARC 90,135,145,320,1,16,14,1
117 PAINT 70,130,1
118 CIRCLE 250,50,20*1.4,20,1
119 PAINT 250,50,1
```

In order for the program to run correctly, one more statement must be changed and two statements added.

The PRINT AT statement used on low-resolution screens must be replaced by the high-resolution TEXT statement, so that the text and the MOB remain on the screen at the same time. If the PRINT AT statement was used, the MOB would be removed from the screen before the text was printed because of changing screen types. (The statements needing a bit-mapped screen—lines 720 through 740—would be executed, then the system would return to a low-resolution screen to perform the PRINT AT statement—line 710.)

And last, the MOB must be cleared off the screen before the program ends to avoid having a distorted MOB displayed on the final screen. The distortion is caused by the screen type reverting to normal—low resolution—when the program ends.

To change the text, view the final screen display for two seconds, and clear the sprite from the screen, add the following statements to the program in memory.

ENTER:

```
710 TEXT 80,80,"GALAXY RULER",1,1,8
730 PAUSE 2
740 RLOCMOB 1,344,120,1,80
```

RUN the program.

The following are important facts to remember when adding sprites to a high-resolution or multicolor screen.

1. When you switch from a high-resolution or multicolor screen to the normal low-resolution screen, the sprites cannot be displayed correctly because bit-mapping is not available. This means that you must think of the program statements used in the context of their screen type.
2. When using both high-resolution and multicolor sprites on either a high-resolution or multicolor screen, all high-resolution or multicolor statements must precede the first DESIGN statement.
3. When structuring your program, be aware of the screen type used by each of the program statements, so that inadvertent switching between screen types does not distort or remove sprites.
4. The HIRES statement and/or the HIRES and MULTI statements are only used for HIRES or MULTI graphics, respectively. These statements do not affect MOBs—either high-resolution or multicolor. Therefore, different colors may be specified for the graphic statements than those specified for MOBs.

Another Programming Option

GRAPHICS holds the decimal constant 53248, the start address of the Video Interface Chip (VIC-II chip). Therefore, for those of you who wish to POKE video chip locations, GRAPHICS plus an offset may be used. For example, to position sprite Ø use GRAPHICS as the x position register and GRAPHICS+1 as the y position register.

Since memory locations used for a MOB on a high-resolution screen end at the start of the VIC-II chip,GRAPHICS minus 64 or a multiple of 64 may be used in the DESIGN statement in place of memory location plus the graphics constant value of 49152. For example, in the current program DESIGN Ø, GRAPHICS-64 and MOB SET Ø,63,1,1,Ø would replace lines 12Ø and 56Ø, and DESIGN 1,GRAPHICS-(64*2) and MOB SET 1,62,7,Ø,1 would replace lines 34Ø and 57Ø

VIII Simons' Sounds

Introduction

Simons' BASIC provides the same ease of use for the Commodore 64's music synthesizing capabilities that it provides for the creation of sprites. This section, like the previous section on sprites, is written to be easily understood by the Commodore BASIC programmer who has not used the POKEs of Commodore BASIC to produce sound. Those of you who have used the capabilities of the SID (Sound Interface Device) chip with Commodore BASIC know that it is capable of producing a wide variety of sounds--but at a great expense of programming effort. Rather than your having to POKE SID memory registers, Simons' BASIC will POKE for you! The only exception is two POKEs for pulse width for use with the pulse waveform. And even this is made easy by providing the SOUND function which holds the decimal address used in the POKES. For both the newcomer to music and the knowledgeable sound user, Simons' BASIC will eliminate much tedious programming.

By the end of Chapter 21, you will at the very least be able to use the music commands to produce music and sound effects at an elemental level. For those of you who wish to take your Commodore 64 musical experience further, I must refer you to the works of others who are more experienced in this field. The Nov/Dec issue of *Commodore Microcomputers* magazine contains five excellent articles both on using the Commodore 64's sound capabilities using Commodore BASIC (there before Simons' sound), and additional music software available for the Commodore 64—making further use of SID. If you're a Commodore 64 owner and not currently a subscriber to this excellent magazine—**unbelievable**—you should get this back issue when you sign-up for a subscription.

I feel that it will be helpful to those of you with limited musical knowledge to begin the review of Simons' Sounds with some basic music facts. Those of you who are knowledgeable musicians will probably want to skip this information.

Basic Music Facts

Musical terms and their meanings:

- Volume refers to loudness.
- Pitch is how high or low a note is, that is, its location on the scale and the octave in which the note is played.
- Note refers to each single musical tone. Notes are named for the letters of the alphabet—ABCDEFG. Every note occurs at higher and lower pitches (higher and lower octaves), repeating the notes over and over.
- A sharp (#) makes the pitch of a note a half step higher, for example, on a piano a white key to a black key.
- A flat (b) makes the pitch of a note a half step lower.
- Octave is group of eight notes, going up or down from the same note—C to C, D to D, E to E, etc. The last note sounds the same as the first, but is higher or lower in pitch since it is the first note in the next octave—up or down.

This information should be helpful to your understanding of the sound commands reviewed in the following chapter. A brief overview of each of these Simons' BASIC commands follows.

Chapter 21

VOL allows you to preset the volume at which the music or sound will be played.

WAVE specifies which of the three Commodore 64 "voices" is to be used, and sets the music voice type. The four available waveforms—**triangle, sawtooth, pulse,** and **noise**—are defined so that you can anticipate the type of sound that each generates.

SOUND assists in entering the pulse width for the pulse waveform.

ENVELOPE defines the "shape," or cycle, of the sound. An introduction to the characteristics of sound vibrations and frequencies is covered as a necessary element to understanding this command. **Attack, Delay, Sustain,** and **Release** are the ENVELOPE parameters that denote the cycle—"shape"— of sound. This cycle controls the volume of the sound as it changes with time—from when the sound begins until it fades to its end.

MUSIC specifies the notes to be generated to create the music or sound effects.

PLAY let's the music begin. The music can either be played while the rest of your program continues to execute or cause the program to pause while the music is played. PLAY can also turn sound off.

Key to Entering Program Listings

MUSIC character strings will be entered as: note (shown by a letter), octave number, function key number. There are no spaces between these entries. The function key number is shown by a lower case f, followed by the function key number, for example, f5, means to press function key 5.

21 Programming Music and Sound

Each of the sound commands—VOL, WAVE, ENVELOPE, MUSIC, and PLAY—must be used together in order to generate music or sound effects on your Commodore 64. The SOUND command will make the two POKE's, necessary only for the pulse waveform, easy to enter. To wet your appetite for Commodore music, this chapter begins with a short demonstration program introducing the group of commands as a whole. So get ready for Music! Music! Music! played in the fourth octave.

The purpose of each command is listed to the left of the program line in which it occurs. As stated in the introduction to this section, the function keys will be designated by a lower case f followed by the function key number.

Command Function	Program Listing
	10 PRINT "[CLEAR]"
Sets the volume at which the music will be played.	20 VOL 15
Sets the voice number and voice type	30 WAVE 1,00100000
Defines the shape of the sound	40 ENVELOPE 1,5,11,10,10
Prints the name of the waveform used, and pauses two seconds.	50 PRINT "TRIANGLE WAVEFORM":PAUSE 2
Uses two character strings, to specify the notes to be	60 A$="[CLR/HOME]1G4f5 A4f5G4f5A4f5G4fA4f5

Command Function	Program Listing
generated.	G4f7E4f5F4f5E4f5F4f5 E4f5F4f5E4f7D4f5E4f5 D4f5E4f5"
	7Ø A1$="D4f5E4f5D4f5E4f5 D4f5C4f5D4f5C4f5D4f5 C4f7[CLR/HOME]G"
Sets music speed and enters the music. Initiates sound, as specified by the WAVE, ENVELOPE, and MUSIC commands	8Ø MUSIC 8,A$+A1$
	9Ø PLAY 1
	1ØØ END

RUN the program.

Now that you have heard Simons' BASIC access the Commodore 64's voice, we'll take a careful look at each of the music commands.

Note: If you are not continuing on in this chapter, SAVE this demonstration program. It will be edited to demonstrate other music commands.

VOL

As the name appropriately states, VOL specifies the volume level (loudness) at which the music or sound following this command will be played. The level of volume specified will not change unless the VOL command is reused in the same program to specify a new volume.

Statement format:
VOL sound level value

The sole parameter, sound level value, allows a range of Ø-15. The number 15 designates the loudest volume, with the volume decreasing as you count down to 1. A value of Ø turns the sound off. The decrease in volume is very gradual, without a noticeable decrease between 15 and 7.

If the value given is less than zero, the program will end with an ILLEGAL QUANTITY ERROR. However, no error message is given for

a value greater than 15, until the value exceeds 255. Only then is an ILLEGAL QUANTITY ERROR generated. Using values having the first two digits of 16 will generate no sound at all. Numbers greater than 16 but less than 255 will generate sound by using the first digit of two-digit numbers greater than 16 and the first two digits of three digit numbers between 100 and 159. A number greater than 159 will generate the loudest volume, no matter what that number is. This lack of upper-limit checking by Simons' BASIC means that you must **be careful not to use a value greater than 15**.

The following changes to the program in memory will demonstrate eight levels of volume, beginning with 15—the loudest volume—and decreasing by steps of two sound levels. The program will end with 1—the lowest volume. The volume used for each repetition of the music will be displayed on the screen immediately before the music begins. You will notice that the decrease in volume is very gradual. This demonstration will give you a good idea of "how loud/how soft" a volume each value represents. This would also be a good time to adjust the volume of your monitor!

Enter the following program changes.

```
5  N=15
85  VOL N:PRINT "VOLUME";N
100 N=N-2:IF N<0 THEN END
110 PAUSE 1:GOTO 85
```

Volume level is checked in line 100 to avoid a negative value and an ILLEGAL QUANTITY ERROR.

If you would rather hear a shorter piece of music for this demonstration, list the program and edit the MUSIC statement, as shown below, to play the major scale—do, re, mi, fa, sol, la, ti, do—rather than Music! Music! Music! Since the edited MUSIC command does not use the music character strings A$ and A1$, they will simply be ignored. However, do not remove these statements (lines 60 and 70) because they will be used again in a later program.

```
80 MUSIC 8,"[CLEAR]1C4f5D4f5E4f5F4f5
G4f5A4f5B4f5C3f5[CLEAR]G"
```

RUN the program for a demonstration of volume levels.

After you have finished the volume demonstration, **remove lines 5, 85, 100, and 110**. If you edited line 80 to play the major scale, list the program and change this statement back to its original form to play Music! Music! Music! as shown below.

```
80 MUSIC 8,A$+A1$
```

This would also be a good time to edit the program to produce the volume of your choice.

Waveforms

Now, if you are not a musician, you must increase your knowledge of the form in which music—or sound—occurs. This means knowing a bit about waveforms before tackling the WAVE command. Waveforms are simply different types of sound waves. Each type of sound wave, because of its particular shape, produces a different effect, as each musical instrument produces a different sound even though they are all playing the same notes. Each of the four Simons' BASIC waveforms are illustrated in Figure 21.1.

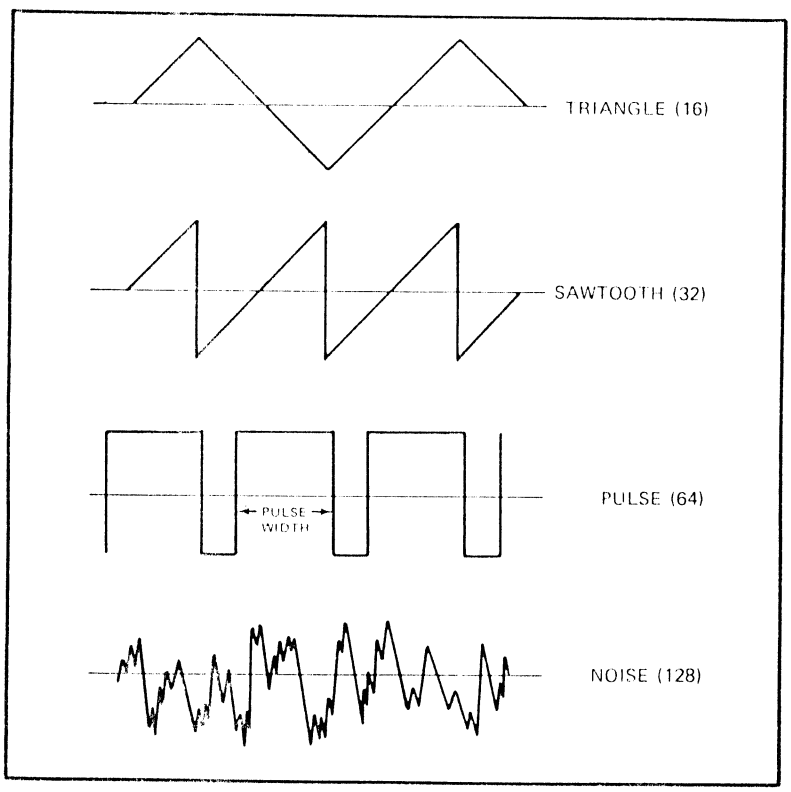


Figure 21.1: Simons' BASIC Waveforms.

The **triangle waveform** is the simplest and purest of the four. It has a mellow, flute-like quality. This wave is smoothly produced by a gradual

increase followed by a gradual decrease in amplitude (loudness). The only abrupt change is at the top of the wave.

The **sawtooth waveform** is rich in harmonics and has a bright, brassy sound. It smoothly rises from minimum to maximum amplitude the same as the triangular waveform. However, then the comparison ends. While the triangular waveform makes a gradual descent in value, the sawtooth waveform drops abruptly in amplitude back to its minimum value. This causes its richer sound.

The **pulse waveform**, a rectangular wave, can produce many sounds by varying the pulse width. A named variation, the pulse/square waveform, produces a sound that is less rich than the sawtooth, but much richer than the triangular waveform, and has a "metallic" or "reedy" sound.

The **noise waveform** is used for sound effects. This waveform generates many frequencies, creating sound without pitch.

WAVE

This command specifies the voice number and the waveform parameters for that particular voice. The waveform parameter also includes the choice of two harmonic structures that can be accessed. They are synchronization and ring modulation, which use two voices. These advanced techniques will be reviewed at the end of this chapter.

Statement format:
WAVE voice number, waveform parameters

The Commodore 64 has three "voices" (oscillators), numbered 1-3. Each voice can be used separately or in combination, and two voices can either be synchronized to produce complex harmonies, or be combined to produce ring modulation to create bell or gong sounds and special effects. Each voice used must have its own WAVE statement.

The waveform parameters are specified as a binary number, with each digit (bit) representing a separate instruction. Each of the eight digits of the binary number must be entered. If you fail to enter all eight digits, you will get the error message NOT BINARY CHARACTER. Figure 21.2 introduces the function of each of the eight digits, from bit 7 down through bit Ø. The binary number is created by assigning a Ø for a

function that is not to be used (turned "off") and a 1 for a function that is to be used (turned "on").

Waveform Parameter Binary Number Function Table

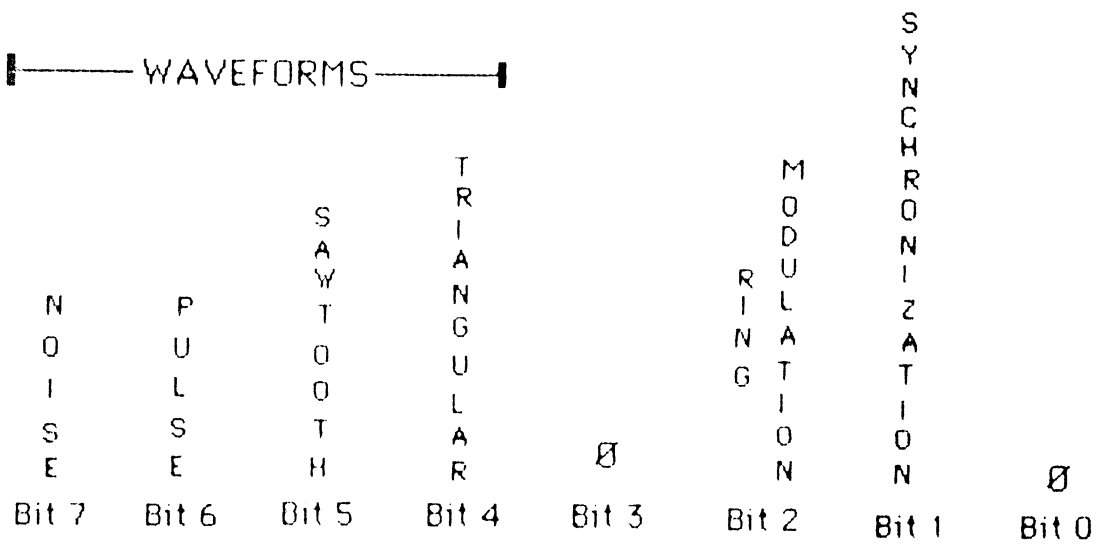


Figure 21.2: Waveform Parameter Binary Number Function Table.

Note: The test bit—bit 3—and the gate bit—bit 0—are always off; that is, always use Ø.

The following is a summary of the function of each bit of the waveform parameter binary number.

Bit 7 A value of 1 initializes the **noise waveform**.

Bit 6 A value of 1 initializes the **pulse waveform.** To generate a pulse waveform, including the pulse/square waveform, the pulse width registers must also be set. The additional statements required are reviewed later in this chapter.

Bit 5 A value of 1 initializes the **sawtooth waveform.**

Bit 4 A value of 1 initializes the **triangular waveform.**

Bit 3 A value of Ø should always be used, since this bit—the Test Bit—is not used in Simons' BASIC.

Bit 2 A value of 1 initializes **ring modulation** by combining the output of two voices.

Bit 1 A value of 1 initializes **synchronization**; a note from one voice can then be synchronized with a note from another voice.

Bit Ø A value of Ø should always be used since Simons' BASIC automatically sets this bit when the PLAY command is executed.

By adding the following statements to the demonstration program, the sawtooth waveform and the noise waveform will be added. Each of the three waveforms—triangle, sawtooth, and noise—will be used to play the notes of Music! Music! Music!. The waveform name will be displayed on the screen just prior to its use in the program.

Program Lines

```
100 X=X+1
110 IF X=1 THEN PRINT "SAWTOOTH WAVEFORM":
WAVE 1,00100000:GOTO 90
120 IF X=2 THEN PRINT "NOISE WAVEFORM": WAVE
1,10000000:GOTO 90
```

This is a valid demonstration of the noise waveform. However, as you will hear, this waveform is **not** used for music! It is used for sound effects and normally requires only a few notes — as shown in the example of gun shots in the *Simons' BASIC User Guide*.

RUN the program and listen carefully to the difference between each of the three waveforms. If you wish to end the program during the demonstration of the noise waveform, press the RUN/STOP and RESTORE keys. The noise waveform is played last for this purpose!

The last waveform to be reviewed is the pulse waveform. As pointed out earlier, the pulse waveform requires a POKE to set each of the two pulse width registers. The POKEs will be familiar to those of you who have programmed sound in Commodore BASIC. However, Simons' BASIC does make this somewhat easier, which brings us to a new Simons' BASIC function.

SOUND Function and the Pulse Waveform

This function holds the decimal constant (54272) equal to the start address of the "SID" chip, the sound and music synthesizer inside the Commodore 64. If you enter PRINT SOUND, the output will be 54272. This function will save you the trouble of remembering, or looking up, this address. You simply use SOUND in place of the decimal address. When setting pulse width for a pulse waveform, you must POKE the value for pulse width into registers 2 and 3.

> The POKEs necessary for a **pulse/square waveform** are:
> POKE SOUND+3,8:POKE SOUND+2,0

SOUND+3 is the address of register 3; the value being entered is 8. SOUND+2 is the address of register 2; the value being entered is Ø.

The waveform binary number used in the WAVE command is not affected. You simply "turn on" bit 6 to designate the pulse waveform, as shown in line 13Ø below. Enter the following statements to the program in memory to replace the noise waveform with the pulse/square waveform.

```
12Ø POKE SOUND+3,8:POKE SOUND+2,Ø
13Ø IF X=2 THEN PRINT "PULSE/SQUARE
WAVEFORM":WAVE 1,Ø1ØØØØØØ:GOTO 9Ø
```

RUN the program for a demonstration of the triangle, sawtooth, and pulse/square waveforms playing Music! Music! Music!.

By changing the value in registers 2 and 3, other pulse waveforms can be generated. Together these registers specify a 12-bit number for the pulse width. The formula for determining pulse width is: PW=Hpw* 256+Lpw, which is Pulse Width=High pulse width (register 3) *256 + Low pulse width (register 2). Register 2 is the low byte of the pulse width (0-255), and register 3 is the high four bits (0-15). By using a different value in the range Ø through 255 for register 2 and/or a different value in the range Ø-15 for register 3, you will change the pulse width and create a different sound quality. For example, the pulse/square waveform has a pulse width value of 2048, 8 (Hpw) * 256 + 0 (Lpw) = 2048 or value of register 3 * 256 + value of register 2.

To demonstrate the change in the sound generated by varying the pulse width, add the following two lines to the demonstration program. This time the value of register 3 is 1 and the value of register 2 is 0, for a pulse width of 256, 1 * 256 + 0.

```
ENTER   14Ø POKE SOUND+3,1:POKE SOUND+2,Ø
        15Ø IF X=3 THEN PRINT "PULSE WAVEFORM
        WITH A PW OF 256":WAVE 1,Ø1ØØØØØØ:GOTO 9Ø
```

RUN the program for a demonstration of four waveforms—triangle, sawtooth, pulse/square, and a pulse waveform.

I'm sure that you will want to experiment with various pulse waveforms of your own.

In addition to waveforms producing a certain quality of sound, the parameters of another command—ENVELOPE—determine the amplitude of a sound as it changes with time. That is, how loud the sound is from the moment you hear it, until it can no longer be heard.

ENVELOPE

The ENVELOPE command defines the shape of each of the sounds to be generated through its four parameters, known as **ADSR**. This controls the rate of the rise and fall and the level of the volume of each musical note. Those of you who are familiar with Commodore BASIC sound know that this term stands for--**attack**, **decay**, **sustain**, and **release**—the four parameters that control the volume of a sound as it changes with time. Attack, delay, and release are units of time, and sustain is the volume. The ADSR envelope is illustrated in Figure 21.3.

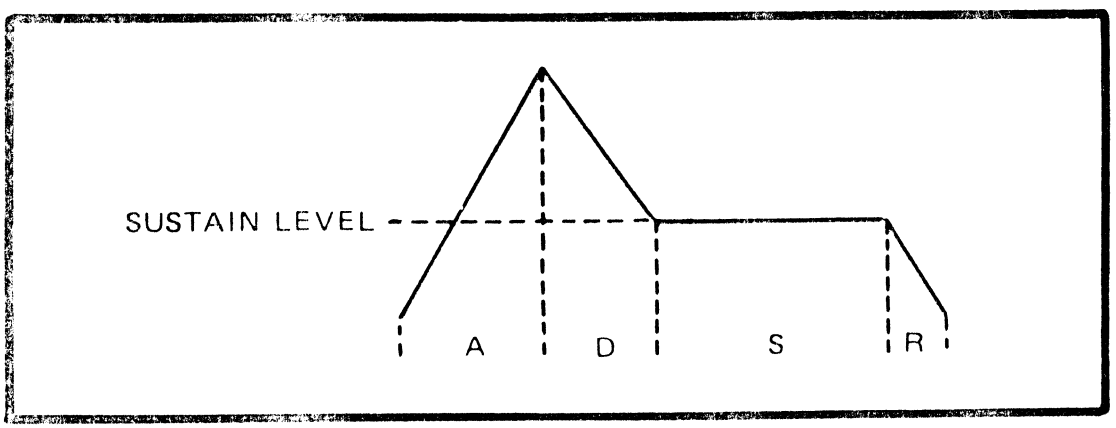


Figure 21.3: ADSR Envelope.

As shown in Figure 21.3, **attack** is the time during which the sound builds from nothing to peak volume. **Decay** is the time it takes for the sound to fall from peak volume to the sustain level. **Release** is the time it takes for the sound to decrease from the **sustain level** to silence. **Sustain** is the volume level at which the note will be held for most of its duration—the mid-range volume.

Statement format:
ENVELOPE voice number, attack, decay, sustain volume, release

The voice number parameter is the same parameter used for the WAVE command. Voice 1, 2, or 3 must be specified. You must have both a WAVE and an ENVELOPE command for each voice, except when using two voices for either ring modulation or synchronization.

The parameters attack, decay, and release are measured in units of one-thousandths of a second—a millisecond (ms). This time duration designates the length of time each of these functions is to be performed. As shown in Figure 21.4, the values Ø-15 each designate a specific time duration in milliseconds. For attack rate, a time duration between 2 ms and 8000 ms may be specified. A time duration between 6 ms and 24000 ms may be specified for both decay and release rate.

Time Value	Attack Rate (ms)	Decay or Release Rate (ms)
Ø	2	6
1	8	24
2	10	48
3	24	72
4	38	114
5	56	168
6	68	204
7	80	240
8	100	300
9	250	750
10	500	1500
11	800	2400
12	1000	3000

Time Value	Attack Rate (ms)	Decay or Release Rate (ms)
13	3000	9000
14	5000	15000
15	8000	24000

Figure 21.4.

The sustain volume parameter uses a value in the range Ø-15. This value has the same amplification—loudness—as in the VOL command. Using a value for sustain volume that is different than the value given in the VOL command changes the volume at which each note will be played *during the sustain period* for the voice specified in the voice number parameter.

Clear computer memory. In the demonstration program that follows, the tune Cabaret will be played using the triangular waveform with a fast attack (16 ms), a medium decay (300 ms), and a slow release (2400 ms). The volume for sustain will be 1Ø. The Simons' BASIC commands illustrated are shown to the left of the program listing.

Commands Illustrated	Program Listing for Cabaret
	1Ø PRINT "[CLEAR]"
WAVE	2Ø WAVE 1,ØØ1ØØØØØ
ENVELOPE	3Ø ENVELOPE 1,2,8,10,11
	4Ø A$="[CLEAR] 1Zf5G4f5A4f7C4f5E4f5G4f7
	[SHIFT G]4f5A4f5G4f5E4f5G4f7Zf3A4f5G4f7
	E4f5G4f7A4f7E4f2Zf5"
	5Ø A1$="A4f5C5f7A4f5C5f7A4f7B4f5G4f7
	E4f2Zf3E4f5G4f7E4f5G4f7A4f7E4f2[CLEAR]G"
VOL	6Ø VOL 8
MUSIC	7Ø MUSIC 7,A$+A1$
PLAY	8Ø PLAY 1
	9Ø END

RUN the music program. Leave your computer on **or** SAVE this program. It will be used again later in this chapter.

MUSIC

This command provides the music sheet that contains the music or sound effect to be played. The note characters correspond as closely as

possible to musical notation. Notes are in the range A-G. C is the first note in each octave, and the sequence of notes is C, D, E, F, G, A, B. To indicate a **sharp**, that note is entered by pressing **SHIFT and** the letter of that **note**. Therefore, a B sharp would be entered as [SHIFT] B. To play a note **flat**, the **preceding note must be entered as a sharp**. *For example*, in entering the major scale—C, D, E, F, G, A, B—to play E flat, D must be entered as a sharp—[SHIFT]D. Music **rests** are indicated by entering the letter **Z**, and are followed by a function key indicating the duration of the rest—the same as for notes.

Statement format:
MUSIC beat, music string

Music string format:
"[CLR/HOME] voice number note string [CLR/HOME] G"

Note string format:
note character, octave number, note duration

The first parameter specifies the duration of one music beat, providing the rhythm of the notes to be played. This value is specified by a number in the range 1-255, with 1 specifying the shortest duration and 255 specifying the longest duration.

The maximum length of a music string is 255 characters. Music strings can be entered as concatenated character strings in order to make use of the maximum length. For example: MUSIC 3Ø, A$+A1$+A2$ specifies the concatenation of three character strings—A$, A1$, A2$—for the music string parameter.

As specified in the note string format shown above, each note is followed by an octave number and a note duration value. The octave in which a note is played is designated by a number Ø-8. The duration of each note is specified by one of eight function keys, each of which represents a specific note duration as shown in Figure 21.5.

Note Duration Table

Function Key	*Note Duration*
f1	One sixteenth of a beat
f3	One eighth of a beat
f5	One quarter of a beat
f7	Half a beat
f2	One beat
f4	Two beats
f6	Four beats
f8	Eight beats

Figure 21.5

Following the last notation—note, octave, note duration—enter [CLEAR] G. This causes the release cycle of each note to be triggered. **Do not forget the G.** If you do, the music will play forever—that is, until you press the RUN/STOP and RESTORE key to end it.

Rests, indicated by the letter Z and followed by a function key for rest duration, may be used anywhere within the note string.

All notes specified in the MUSIC command are played using the parameters specified in the WAVE and ENVELOPE commands for the same voice number.

LIST the program in memory—Cabaret—or LOAD the program now to review the MUSIC strings used for playing Cabaret. A$ includes a G sharp, indicated by [SHIFT G]. Both music strings contain rests—indicated by Z—of either one eighth of a beat or one quarter of a beat. After studying these entries, you should be able to enter your own music. You only need to remember:

> Begin the first music string with [CLEAR] followed by the voice number. Thereafter, the entry for each note is: letter of the note, octave number, note duration. The final music string **must** end with [CLEAR] and the letter G.

PLAY

This is the command that activates the MUSIC, letting the sounds in the music strings be executed.

Statement format:
PLAY n

The parameter n specifies how the music will be played in relation to the program as a whole. The values used to designate each function are shown in Figure 21.6.

PLAY Functions

Number Parameter	*Function Performed*
Ø	Turns the music off.
1	Plays the music, then proceeds with the program.
* 2	Plays the music while continuing to execute the program.

Figure 21.6.

* PLAY 2 **cannot** be used in conjunction with high-resolution or multicolor graphics.

The Cabaret music program will now be used to demonstrate the two values of the PLAY command that have not yet been used—PLAY 2 and PLAY Ø. Line 8Ø edits the PLAY statement to continue playing the music while executing the rest of the program. The added FOR NEXT loop will print the value of I while the music plays.

```
ENTER:    8Ø PLAY 2
          9Ø FOR I=1 TO 35Ø:PRINT "I=";I:NEXT I
```

The music will end before the program does, since the music is only played through once. RUN the program.

If the program ends before the music is finished playing, the note that is being executed will be held until you stop program execution using the RUN/STOP and RESTORE keys. The PLAY command will not be turned off by the program ending or by executing PLAY Ø. The entire music string must be played through once so that the letter G, which causes the release cycle of each note to be triggered, can be reached. Therefore, there must be a sufficient number of program statements to be executed during the playing of the music. To demonstrate this, LIST the program, edit line 9Ø to shorten the loop by changing the upper

limit in the FOR statement to 1ØØ and RUN the program again. You will have to press the RUN/STOP and RESTORE keys to end the holding note.

To turn the music off, simply replace line 9Ø, as shown below.

ENTER: 9Ø PLAY Ø

Now the program will run—silently. RUN the program.

Advanced Techniques—Ring Modulation and Synchronization

You can create more complex harmonic structures through synchronization or ring modulation of two voices.

Ring modulation is created by varying the frequency of one voice against a static frequency of a second voice. This combining of the two voices produces a wide range of non-harmonic structures for use in creating bell or gong sounds and special effects.

The **variable frequency (multiple note) voice must be a triangular waveform.** This triangular waveform is then replaced by a modulated combination of the output of this voice and the voice specified as the static frequency (one note) voice. The static frequency voice **cannot** use the noise waveform. As shown in Figure 21.7, the voice number chosen to output the variable frequency determines the voice number that **must** be used for the static frequency.

Ring Modulation Voice Number Table

Variable Frequency Voice Use Triangular WAVE	*Static Frequency Voice Use Triangular or Sawtooth WAVE*
1	3
2	1
3	2

Figure 21.7.

Ring modulation is initialized when bit 2 and the triangle waveform are turned on in the WAVE command used by the variable frequency voice. The WAVE command binary number would be: ØØØ1Ø1ØØ. An

ENVELOPE command is only provided for the variable frequency voice, since this is the voice that will output the combined sound of the two voices.

The following ring modulation demonstration program produces an imitation of a clock chime. Ring modulation replaces the triangle waveform output with a combination of voices 1 and 3. Note C7 of voice 1 and note B4 of voice 3 are combined. Voice 1 uses the triangle waveform and voice 3 uses the sawtooth waveform.

Clear computer memory and enter the following program. The Simons' BASIC commands are shown to the left of the program listing.

Command Format	Program Listing
	10 REM CLOCK CHIME
VOL	20 VOL 15
WAVE	30 WAVE 1,00010100
ENVELOPE	40 ENVELOPE 1,0,9,0,0
WAVE	50 WAVE 3,00100000
	60 FOR I=1 TO 5
MUSIC	70 MUSIC 20,"[CLEAR]1C7[CLEAR]
	3B4f3[CLEAR]1[CLEAR]C[CLEAR]3
	[CLEAR]C[CLEAR]"
PAUSE	80 PAUSE 1
PLAY	90 PLAY 1
	100 NEXT I
	110 END

RUN the program for a demonstration of clock chimes.

Voice 1—the variable frequency voice—uses only one note in this particular program, but can be used to output multiple notes (as many different notes as you like). Each note will be combined with the one static frequency note. In coding the MUSIC command you must alternate each variable frequency note followed by [CLEAR] with the static frequency note so that they may be properly combined for output through the variable frequency voice. For example, in 1C5[CLEAR]3B4[CLEAR] 1D6[CLEAR]3B4[CLEAR]1E7[CLEAR]3B4 the variable notes C5, D6, and E7 would, each in turn, be combined with the static frequency note—B4. You should also end the music string with C[CLEAR] for both voices, that is, 1[CLEAR]C[CLEAR]3[CLEAR]C[CLEAR], to set the control byte to zero to stop sound.

The last technique to be reviewed is **synchronization**, which plays

multiple notes from one voice synchronized with one steady note from a second voice.

By playing one steady note (static frequency) with one voice and multiple notes (variable frequency) with a second voice at the same time, a wide range of harmonies can be produced. For the best effect, the one steady note of the static frequency voice should be lower than the lowest note played by the variable frequency voice. As in ring modulation, the voice number chosen to output the variable frequency notes determines the voice number that you **must** specify for the static frequency. The static voice used for each variable frequency voice is the same for synchronization and for ring modulation, as shown in Figure 21.8.

Synchronization Voice Number Table

Variable Frequency Voice	*Static Frequency Voice*
1	3
2	1
3	2

Figure 21.8.

Synchronization is initialized when both bit 1 and any waveform bit are "on" in the WAVE command of the variable frequency voice. Again, an ENVELOPE command is only required for the variable frequency voice since this voice will output the combined sounds of both voices, and the notes are again entered for the two voices in alternating fashion. The string should again end by turning off each voice—C[CLEAR].

The following demonstration program uses synchronization to produce the sound of a very annoying mosquito.

Clear computer memory and enter the following program. The Simons' BASIC commands are shown to the left of the program listing.

Command Format	Program Listing
	10 REM HUMMING MOSQUITO
VOL	20 VOL 15
WAVE	30 WAVE 1,00010010
ENVELOPE	40 ENVELOPE 1,13,11,15,12
WAVE	50 WAVE 3,00010000

```
MUSIC          60 MUSIC 100,"[CLEAR]1G6[CLEAR]
               3A4f3[CLEAR]1[CLEAR]C[CLEAR]3
               [CLEAR]C[CLEAR]"
               70 FOR I=1 TO 5
PLAY           80 PLAY 1
PAUSE          90 PAUSE 1
               100 NEXT I
               110 END
```

RUN the humming mosquito demonstration.

Needless to say, with the capabilities of the Commodore 64's SID chip and the ease of using the Simons' BASIC language, the demonstration programs in this chapter barely scratch the surface of your computer's musical capabilities. Remember, the SID chip's greatest value lies in creating sounds that cannot easily be made by real instruments. Only a great deal of experimentation with these music commands can begin to show the multitude of musical possibilities.

When experimenting with the above humming mosquito program, I found that using CLEAR for the two voices **and** for the two notes generated a more realistic humming. The sound fades, rather than ends, before the sound is repeated. If you would like to try this, edit the MUSIC statement as follows:

```
60 MUSIC 100,"[CLEAR]1G6[CLEAR]3A4f3[CLEAR]1[CLEAR]
G[CLEAR]3[CLEAR]A[CLEAR]"
```

For those musicians who wish to experiment further, here is another tip from David Simons that is not included in your User Guide. For synchronization, you can use [CLEAR]T to toggle (flip-flop) the synchronization bit. That is, you can reverse bit 1 from on to off, or from off to on.

IX Leaving the Keyboard

Introduction

There are four functions to assist in communicating with your computer without using the keyboard. All of these will more than likely be used by gaming programmers. The JOY function eliminates PEEKing for the direction in which the joystick—the most popular of the game controllers—is pointing. Similarly, the POT function will give you the current rotation of either game paddle. These two video game controllers used to be the ultimate in non-keyboard game playing. Now, of course, there's also the light pen, and David Simons didn't forget this peripheral either! The functions for returning the exact location at which the light pen is pointing are PENX and PENY.

The following is a brief overview of each of these functions.

Chapter 22

JOY allows the programmer to access the direction in which the joystick is pointing and to determine if the fire button has been pressed.

POT allows the programmer to access the amount of rotation of either of two game paddles.

PENX stores the horizontal location (x coordinate) at which the light pen is pointed.

PENY stores the vertical location (y coordinate) at which the light pen is pointed.

Key to Entering Program Listings

There are no additional key conventions used in this section.

22 Joystick, Paddle, and Light Pen Read Functions

The following is a review of the four read functions supplied for the Simons' BASIC cartridge.

JOY

This function returns a value indicating either the current joystick position or use of the fire button.

Function format:
JOY

The value returned for each joystick position and for the fire button are shown in Figure 22.1.

The following program will print the value of JOY and name the position in which the joystick is pointing. The first position, of course, will be CENTER. There will then be a pause of two seconds before the computer will "read" another position. You may use AUTO 10,10 for automatic line numbering.

ENTER:
```
10 PRINT "[CLEAR]"
20 X=0
30 PRINT JOY,
40 IF JOY=0 THEN PRINT "CENTER":GOTO 140
```

```
50 IF JØY=1 THEN PRINT "NORTH":X=JOY:
GOTO 14Ø
6Ø IF JOY=2 THEN PRINT
"NORTHEAST":X=JOY:GOTO 14Ø
7Ø IF JOY=3 THEN PRINT "EAST":X=JOY:
GOTO 14Ø
8Ø IF JOY=4 THEN PRINT
"SOUTHEAST":X=JOY:GOTO 14Ø
9Ø IF JOY=5 THEN PRINT "SOUTH":X=JOY:
GOTO 14Ø
1ØØ IF JOY=6 THEN PRINT
"SOUTHWEST":X=JOY:GOTO 14Ø
11Ø IF JOY=7 THEN PRINT "WEST":X=JOY:
GOTO 14Ø
12Ø IF JOY=8 THEN PRINT
"NORTHWEST":X=JOY:GOTO 14Ø
13Ø IF JOY=128 THEN PRINT "FIRE BUTTON
PRESSED":X=JOY:GOTO 14Ø
14Ø PAUSE 2:IF JOY=X GOTO 14Ø:ELSE:
GOTO 2Ø
```

Before running the program, insert a joystick into **control port 2.** This is the second port from the front on the right side of the computer. JOY does not function properly when control port 1 is used. The example program in your *Simons' BASIC User Guide* will not run from control port 1 either.

RUN the program. When you're tired of testing your joystick, hold down the RUN/STOP key to end.

Note: It may be my particular joystick, but I have a problem with programs using the joystick "hanging" after being run a few times. The only remedy is to turn off the computer. Therefore, I would recommend saving programs that use a joystick immediately after they are written, just in case the joystick problem isn't mine alone!

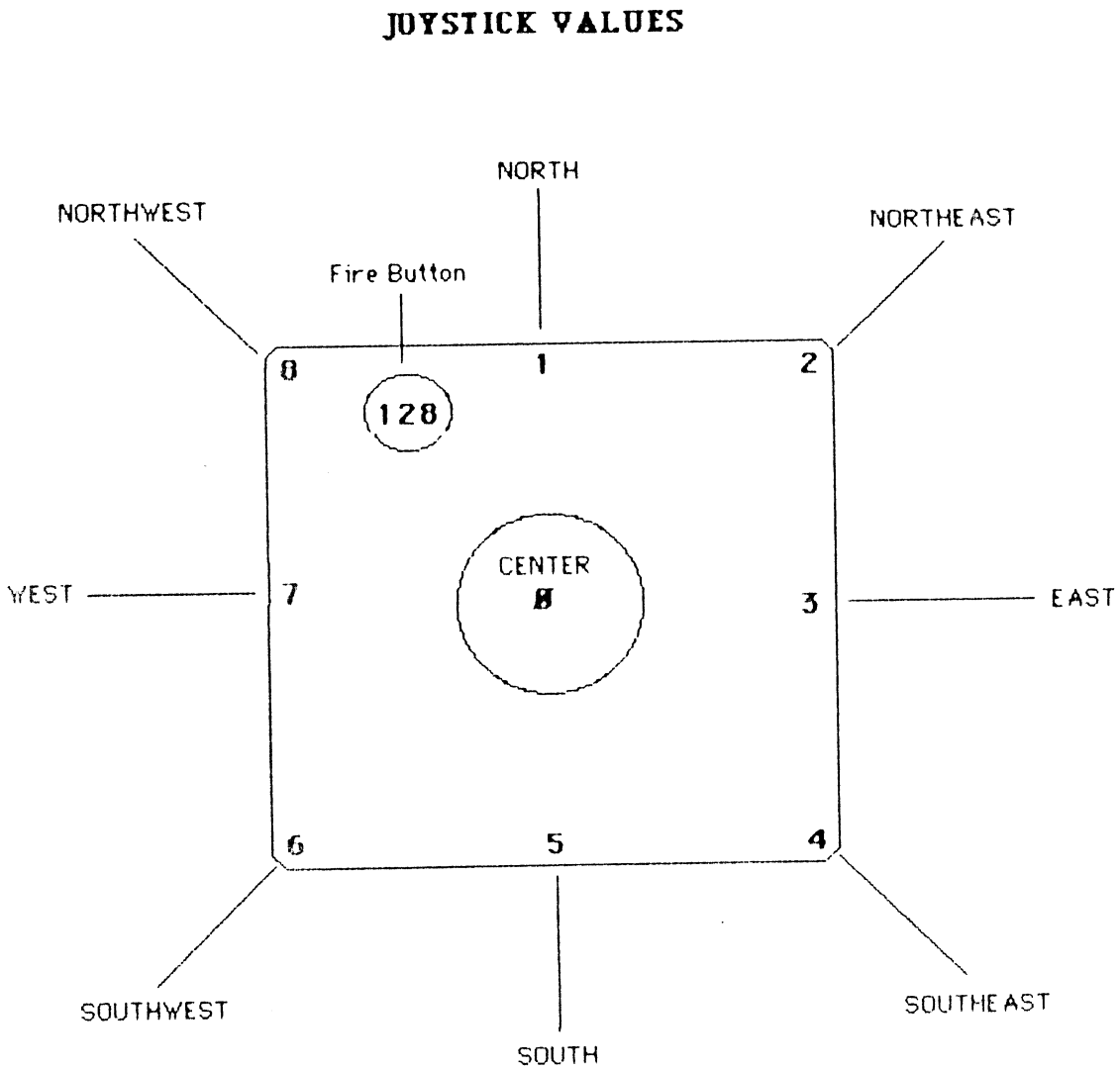


Figure 22.1: Picture of a joystick, with values noted.

POT

The POT function returns a value in the range Ø-255, which indicates the current paddle knob rotation. The number Ø indicates that the paddle is turned all the way to the right, and the number 255 indicates complete rotation to the left.

Function format for the left paddle:
POT(Ø)

Function format for the right paddle:
POT(1)

The following brief program will show you the number position of the left paddle as it is rotated. There is a two-second pause between paddle rotation value checks.

ENTER:
```
1Ø REM LEFT PADDLE=POT(Ø); RIGHT
PADDLE=POT(1)
2Ø PRINT POT(Ø)
3Ø PAUSE 2
4Ø GOTO 2Ø
```

Before running the program, insert your game paddles into **control port 1**. This is the first port from the front on the right side of the computer. This program does not function properly when using control port 2.

RUN the program. Press the RUN/STOP key to end the program.

Note: I only get different readings in about the first half of a full right-to-left rotation on both paddles. Again, it may only be my paddles!

For those of you who may not know, a light pen is a hand-held, pen-shaped sensor used to point to objects or characters on a video monitor. The computer can then be programmed to read the position of this beam of light and use this location data in the program's decision making. Light pen readings tell you the position on the screen—x,y coordinates—at which the light pen is pointing.

A light pen must only be inserted in **control port 1**. This is the second control port from the front, on the right side of the Commodore 64 computer.

PENX

This function returns the horizontal position—column position—of the light beam. That is, the x coordinate. The number returned is in the range Ø-319, since it reads the position using bit mapping. As you know, the bit- mapped screen contains 32Ø positions, Ø-319.

Function format:
PENX

The value of PENX **must be read before** the value of PENY—the y coordinate.

PENY

This function returns the y coordinate—vertical, or row, position—of the light beam. Again, remember that this function is reading the screen in bit mapping and so returns a number between Ø and 199. The bit-mapped screen is 2ØØ pixels from top to bottom.

Function format:
PENY

The value of PENY **must be read after** the value of PENX is read.

If you have a light pen or would like to add this tool to your programming adventures, PENX and PENY will be a great help in incorporating it in your programs. You only need to read PENX and PENY to know where the program user is pointing, and base you programming decisions on specific coordinates.

Appendix A

Simons' BASIC Command Reference Listing

USE: **C:** COMMAND **S:** STATEMENT **F:** FUNCTION

Low-Resolution Mode Commands

AT(column,row)"character string" **F**
AT(column,row)string variable name
This PRINT function positions text at a user-specified screen location.

@ **S**
The at sign begins each line of a user-defined character grid.

AUTO starting line number,increment **C**
Provides computer-generated line numbers.

BCKGNDS sc, b1 color,b2 color,b3 color **C,S**
Sets the character background color of non-graphic characters.

BFLASH speed,c1,c2 **C,S**
Used to flash the screen border at the specified speed, alternating between the two colors specified. Unit of speed is in the range 1-255, with each unit representing one-sixteenth of a second.

BFLASH Ø C,S
Turns the flashing of the screen border off.

CALL procedure name S
Transfers program control to the named open-ended procedure.

CENTRE"character string" C
CENTREstring variable name C
CENTRE"character string":PRINT S
CENTREstring variable name:PRINT S
Centers text on a screen line without the user specifying text length.

CGOTO numeric expression S
Used to branch to the line number obtained as a result of the evaluation of numeric expression.

COLD C
Clears memory and returns the system to the start of Simons' BASIC.

COLOUR border color number,screen color number C,S
Sets the border and screen colors.
Note: Commodore colors and their associated numbers are shown in Figure 13.1, page 108.

CSET Ø C,S
Specifies the use of the uppercase/graphics character set.

CSET 1 C,S
Specifies the use of the upper/lowercase character set.

DELAY print speed C
Print speed range is 1-255: minimum speed is 1; maximum speed is 255. Used with the SHIFT key to control listing speed.

DESIGN 2,$E000+character poke code***8** S
Specifies the location of the Commodore character to be replaced.

DIR"$ **C**
DIR"$:character(s)*
DIR"$:?character(s)
DIR"$:?character(s)*
Lists all or specific files of a disk directory. Options are, in the order listed above: list all; list using beginning character(s) specified; list using specific character(s) in a specific location of the filename; and list using a combination of beginning and specific location filename characters.

DISAPA:statements **S**
Used to specify code to be hidden.

DISK"operation" **C**
Provides simplified disk entry, eliminating the need to specify a logical file number, device number, and secondary address.

DISPLAY **C**
Lists the function key assignments.

DOWNW starting row,starting column,ending column,ending row **C,S**
DOWNB starting row,starting column,ending column,ending row
W designates wraparound; **B** designates blanking.
Scrolls the specified screen area down.

DUMP **C,S**
Displays the value of all non-array variables.

DUP("character string",duplication number) **C,S**
DUP(string variable name,duplication number)
Duplicates the specified character string the designated number of times, creating an expanded string.

END PROC **S**
The end statement for a closed procedure, functioning the same as the RETURN statement used with GOSUB in BASIC.

EXEC procedure name **S**
Transfers program control to the named closed procedure.

FCHR row,column,width,depth,character code **C,S**
Specifies character to be printed in a specific screen area. **Note:** FCOL must be used with this command.

FCOL row,column,width,depth,color **C,S**
Used to change the print color of characters.

FETCH "control character",character limit,variable name **S**
Limits keyboard input by controlling the type (alphabetic or numeric) and number of characters accepted as input. The control characters are shown in Figure 8.2, page 63.

FILL row,column,width,depth,character code,color number **C,S**
Fills the specified screen area with the designated character in the specified color.

FINDcode **C,S**
FIND"character string"
Locates and displays the line numbers of each occurrence of the specified code or character string.

FLASH color number **C,S**
FLASH color number,speed
Flashes character background color by alternating between normal and reverse field. The range for speed is 1-255, with each number representing approximately one-sixteenth of a second. Default speed is four seconds.

GLOBAL **S**
Used to restore the original values of all variables previously defined as LOCAL.

HRDCPY **S**
Used to print hard copy of a low-resolution screen on a serial printer.

IF condition **THEN** statement(s):**ELSE**:statement(s) **S**
Used to test both consequences, true and false, of the IF condition and address both on the same statement line.

ON **INKEY** GOTO statement line list:GOTO line number **S**
ON **INKEY** GOSUB statement line list:GOTO line number
Accepts function key input (1-8) as the value of INKEY.
Note: The GOTO statement is needed to pause the program until a function key is pressed.

INSERT("sub string","main string",insert position of sub-string) **S**
INSERT(string variable name,string variable name,insert position of sub-string variable)
Inserts one character string into another, creating an expanded character string.

INST("sub-string","main string",insert position of sub-string) **S**
INST(string variable name,string variable name,insert position of sub-string variable)
Used to overwrite a portion of a character string (main string) with another string (sub-string), creating a new character string.

INV row,column,width,depth **C,S**
Used to reverse character background; reverse field to normal or normal to reverse field.

KEY function key number,"command" **C**
KEY function key number,"command" + CHR$(13)
Used to assign BASIC or Simons' BASIC commands to the function keys. Including a carriage return (second format) is optional.
Note: The key presses used for the 16 function key assignments is shown in Figure 4.1, page 32.

LEFTW starting row,starting column,ending column,ending row **C,S**
LEFTB starting row,starting column,ending column,ending row
W designates wraparound; **B** designates blanking.
Used to scroll the specified screen area to the left.

LOCAL variable list **S**
Stores the values of *previously defined* variables (assumed to be GLOBAL) and clears the values for reuse.
Note: LOCAL must be the last statement on a multiple-statement line.

LOOP:loop statement(s):**EXIT IF** condition:**END LOOP** **S**
Provides multiple exit condition testing within the loop. Any number of EXIT IF condition statements may be used anywhere within the loop.

MEM **S**
Moves the Commodore character set from ROM into RAM so that the characters may be replaced with user-defined characters.

MERGE "program name",device number **C**
Adds a program from a storage device to the end of the program currently in memory.

MOVE row,column,width,lines,receiving row,receiving column **C,S**
Copies the specified portion of the screen, displaying the duplicate at the given screen location.

NO ERROR **S**
NO ERROR:PRINT ERRN;ERRLN
Turns off error trapping. (See ON ERROR.) Second format also prints the error number and the line number in which the error occurred.

OFF **C,S**
Turns character background flashing off.

OLD **C**
Restores most recent program cleared from memory by NEW, if no new program line has been entered.

ON ERROR:GOTO line number of NO ERROR statement **S**
Used to trap any one of 19 specific programming errors.

ON KEY "characters",:GOTO line number **S**
ON KEY string variable,:GOTO line number
Accepts keyboard input of only the characters specified as the condition for the branch.
Note: To pause the program until a valid character is input, this statement must be followed—on a separate line—by a GOTO branch to the ON KEY statement.

OPTION 1Ø C
Used to highlight (using reverse-field printing) all Simons' BASIC keywords in a program listing.

OPTION Ø C
Used to turn off highlighting.

PAGE number of lines C
Controls the number of screen lines printed by LIST.

PAGE Ø C
Turns paging off.

PAUSE number of seconds C,S
PAUSE "message",number of seconds
Prints optional message, if present. Halts program execution for the specified number of seconds.

PLACE("sub-string","main string") S
PLACE(string variable name,string variable name)
Returns the starting location of a sub-string within a main string.

PROC procedure name S
Used to name a closed or open-ended procedure.

RCOMP:statement(s)**:ELSE:**statement(s) S
Re-executes the most previous IF condition test using the RCOMP and ELSE parameters.

RENUMBER starting line number, increment C
Renumbers all line numbers of the program in memory.
Note: Does not renumber line number references in program statements.

REPEAT:loop statement(s)**:UNTIL** condition C,S
Performs a program loop based on a condition test at the end of the loop.

RESET data statement line number S
Used to READ a specific portion of data by branching directly to a specific DATA statement line.

RESUME **S**
Causes a branch back to the previously executed ON KEY statement.

RETRACE **C**
Resumes tracing.

RIGHTW starting row,starting column,ending column,ending row **C,S**
RIGHTB starting row,starting column,ending column,ending row
W specifies wraparound; **B** specifies blanking.
Used to scroll the specified screen area to the right.

SCRLD 2,8,2,"name" **C**
Redisplays a screen saved on disk.

SCRLD 1,1,Ø,"name" **C**
Redisplays a screen saved on tape.

SCRSV 2,8,2,"screen name**,S,W**" **S**
Saves a low-resolution screen display to disk.

SCRSV 1,1,1,"screen name" **S**
Saves a low-resolution screen display to tape.

SECURE Ø **C**
Secures all DISAPA: code so that it is no longer listable.
Note: Because this command is not reversible, it is advisable to SAVE a program before using SECURE.

TRACE 1Ø **C,S**
Displays each line number as it is executed.

TRACE Ø **C,S**
Turns off tracing.

UPW starting row,starting column,ending column,ending row **C,S**
UPB starting row,starting column,ending column,ending row
W designates wraparound; **B** designates blanking.
Scrolls the specified screen area up.

USE "#.#,""character string" **C,S**
USE "#.#," string variable name
USE "text#.#text","character string"
USE "text#.#text",string variable name
Used to format numeric data. Text may optionally preceed and/or follow the hash marks.
Note: PRINT must follow a USE command when a carriage return is required.

Mathematic Functions

DIV(x,y) **F**
Returns the quotient resulting from the division of two positive, integer numbers.

EXOR(n,n) **F**
Performs an exclusive OR (XOR) between two numbers.

FRAC(numeric expression) **F**
Returns the fractional part of a number resulting from the evaluation of the numeric expression.

MOD(x,y) **F**
Returns the remainder of a division of two positive, integer numbers.

%binary number **F**
Converts the specified binary number to decimal form.

$hexadecimal number **F**
Converts the specified hexadecimal number to decimal form.

Graphics Mode Commands

Plotting parameter definitions:

x=starting column	x1=ending column	xr=horizontal radii
y=starting row	y1=ending row	yr=vertical radii

High-Resolution Mode Plot Types

Plot Type	Function
Ø	Turns off plotting, (clears what has been plotted, or leaves cleared what has not been plotted).
1	Turns on plotting.
2	Reverses plot types Ø and 1. If plotting was on, plotting is cleared (turned off). If plotting was off, plotting is turned on.

Multicolor Mode Plot Types

Plot Type	Function
Ø	Clears a dot.
1	Uses plotting color specified in the color 1 parameter.
2	Uses plotting color specified in the color 2 parameter.
3	Uses plotting color specified in the color 3 parameter.
4	Reverses the order of the plotting color used: Plot color Ø changes to color 3 Plot color 1 changes to color 2 Plot color 2 changes to color 1 Plot color 3 changes to color Ø

ANGL x,y,angle,xr,yr,plot type **C,S**

Plots the radius of a circle at the specified screen location. **Note:** To correctly plot the radius of a circle on a video monitor, an adjustment factor must be used. The recommended adjustment factor is 1.4. For a high-resolution screen, multiply the horizontal radii by this adjustment factor. For a multicolor screen, multiply the vertical radii by this adjustment factor.

ARC x,y,sa,ea,i,xr,yr,plot type **C,S**

Plots a portion of the circumference of a circular shape at the specified screen location. sa=starting degree of angle; ea=ending degree of angle; i=increment of plotting.

Note: To correctly plot an arc on a video monitor, an adjustment factor must be used. The recommended adjustment factor is 1.4. For a high-resolution screen, multiply the horizontal radii by this adjustment factor. For a multicolor screen, multiply the vertical radii by this adjustment factor.

BLOCK x,y,x1,y1,plot type **C,S**
Plots a rectangle or square at the specified screen location, and fills it with the current plotting color.
Note: To plot a perfect square on a video monitor, an adjustment factor must be used. The recommended adjustment factor is 1.4. For a high-resolution screen, the width is multiplied by the adjustment factor. For a multicolor screen, the depth is multiplied by this adjustment factor.

CHAR x,y,poke code,plot type,height **C,S**
Prints a single character on a graphics screen. Values for the height parameter are shown in Figure 15.3, page 143.

CIRCLE x,y,xr,yr,plot type **C,S**
Plots a circular or elliptical shape on a graphics screen.
Note: To plot a perfect circle on a video monitor, an adjustment factor must be used. The recommended adjustment factor is 1.4. For a high-resolution screen, multiply the horizontal radii by this adjustment factor. For a multicolor screen, multiply the vertical radii by this adjustment factor.

COPY **S**
Produces a hard copy of a graphic screen on a serial printer.
Note: If an adjustment factor was used in any of the plotting commands, it must be removed in order to print a correctly proportioned graphic.

CSET 2 **C,S**
Recalls the previously displayed graphic screen.

DRAW "design instructions",x,y,plot type **C,S**
Used to create a graphic design, pixel by pixel, to be displayed at the specified screen location.
Note: The ROT command must be used to display the design.

HI COL C,S
Reverts to the colors specified in the most recently executed HIRES statement.

HIRES plotting color,background color C,S
Initiates the high-resolution, bit-mapped screen, and sets the screen foreground and background colors.

LINE x,y,x1,y1,plot type C,S
Plots a straight line—horizontal, diagonal, or vertical—as specified by the beginning and ending coordinates.

LOW COL color1,color2,color3 C,S
Specifies new graphics plotting colors, replacing those specified in HIRES or MULTI.
Note: In high-resolution mode color1 replaces the plotting color specified in the HIRES command, and color2 adds a plotting background color. Color3 must be specified, but is ignored.

MULTI color 1,color 2,color 3 C,S
Initializes the multicolor screen and specifies the three plotting colors to be used.
Note: The HIRES command **must** precede this command.

NRM C,S
Returns the Commodore 64 to low-resolution mode from the graphics mode.

PAINT x,y,plot type C,S
Fills any closed graphic shape with the current plotting color.
Note: When using the multicolor mode, this command may be reused to repaint the graphic shapes with new colors, as specified in plot type.

PLOT x,y,plot type C,S
Fills—"turns on"—one pixel at the specified screen location.

REC x,y,width,depth,plot type C,S
Plots a rectangle or square using the specified screen location.
Note: To plot a perfect square on a video monitor, an

adjustment factor must be used. The recommended adjustment factor is 1.4. For a high-resolution screen, the width is multiplied by the adjustment factor. For a multicolor screen, the depth is multiplied by this adjustment factor.

ROT rotation number,size **C,S**
Displays the design specified in DRAW. Degree of rotation and size specifications are given in Figure 16.2, page 170.

numeric variable name=**TEST**(x,y) **C,S**
Returns the status—"on or off"—of the pixel at the specified location. A value of 0 specifies "off" (no plotting) and a value of 1 specifies "on."

TEXT x,y,"[CTRL A]character string",plot type,height,spacing **C,S**
TEXT x,y,"[CTRL B]character string",plot type,height,spacing
Prints character strings on a graphics screen in both upper- and lowercase, in varying heights and with varying spacing between characters. CTRL A designates upper case; CTRL B designates lower case. Upper case is the default. The values used for the height and spacing parameters are shown in Figure 15.3, page 143.

Sprite Commands

@ **S**
Used to set up the design grid for a MOB. The grid for a high-resolution MOB is 24 pixels wide by 21 lines deep. For a multicolor MOB the grid is 12 pixels wide and 21 lines deep.

IF **CHECK**(MOB number,MOB number)=0 THEN statement(s) **S**
Used for collision testing between two MOBs.

IF **CHECK(0)**=0 THEN statement(s) **S**
Used for collision testing between any sprite and screen data.

CMOB color B,color D **S**
Specifies the two additional colors available for a multicolor MOB.

DESIGN resolution,starting memory location **S**
DESIGN resolution,starting memory location + graphic constant value
Reserves 64 bytes of memory for a high-resolution or multicolor MOB. The first format shown is used for the low-resolution screen. The second format is for the high-resolution screen, using decimal 49152 or hexadecimal $C000 as the graphic constant value.

DETECT 0 **S**
Used to detect sprite-to-sprite collisions.

DETECT 1 **S**
Used to detect sprite and screen data collisions.

GRAPHICS **F**
Holds the value 53248, the start address of the Video Interface Chip.

MMOB number,x,y,x1,y1,expansion,speed **S**
Displays, with or without movement, a high-resolution or multicolor MOB at the screen location, size, and speed specified. Speed of movement is in the range 1-255, with 1 specifying the fastest speed and 255 the slowest speed. The display sizes that can be specified by the expansion parameter are shown in Figure 19.2, page 197.

MOB OFF number **S**
Clears the specified MOB from the screen.

MOB SET number,block,color,priority,resolution **S**
Initializes a high-resolution or multicolor MOB.

RLOCMOB number,x1,y1,expansion,speed **S**
Moves an on-screen MOB to a new screen location.

Music Synthesis Commands

ENVELOPE voice number,attack,decay,sustain volume,release **S**
Defines the shape of the sound to be generated.
Values used for attack, decay, and release rates are shown in Figure 21.4, page 223.

MUSIC beat,music string **S**
Specifies the music to be generated by a specific voice(s).
Note: Function key specifications for note duration are shown in Figure 21.5, page 226.

PLAY n **S**
Activates the music. The number Ø turns the music off, 1 plays the music and proceeds with the program, and 2 plays the music while continuing to execute the program.
Note: PLAY 2 cannot be used in conjunction with high-resolution or multicolor graphics.

SOUND **F**
Holds the value 54272, the start address of the SID chip.

VOL sound level value **S**
Specifies the volume at which the music or sounds will be played. The sound level range is Ø-15; maximum volume is 15, minimum volume is 1, and Ø turns the sound off.

WAVE voice number,waveform parameters **S**
Specifies the waveform parameters for the designated voice. The waveform parameters are shown in Figure 21.2, page 219.

Video Game Controller Functions

JOY **F**
Returns a value designating the current joystick position or use of the fire button. Joystick position values are given in Figure 22.1, page 236.

PENX **F**
Returns the horizontal position (x coordinate) at which the light pen is pointed.
Note: This value must be read before the value of PENY can be read.

PENY **F**
Returns the vertical position (y coordinate) at which the light pen is pointed.

Note: This value must be read after the value of PENY is read.

POT(Ø) **F**

Returns a value in the range Ø-255, indicating the current knob rotation of the left paddle.

POT(1) **F**

Returns a value in the range Ø-255, indicating the current knob rotation of the right paddle.

Appendix B

List of Reference Tables and Screen Grids

Figure Number	Title	Page Number
2.2	Errors Trapped by ON ERROR	21
4.1	Accessing the 16 Function Keys	32
8.1	ON KEY Valid Characters	60
8.2	FETCH Control Characters	63
13.1	Commodore Colors—Numbers and Key Presses	108
S5.1	High-Resolution Screen Layout	135
15.1	High-Resolution Plot Types	138
15.3	TEXT Height and Spacing Values	143
16.1	Design Instructions for the DRAW Command	169
16.2	Rotation Instructions	170
S6.1	Low-resolution, High-resolution and Multicolor Screen Grid	178

Figure Number	Title	Page Number
18.1	Multicolor Screen Grid	180
18.2	Multicolor Plot Types	181
19.1	Sprite Screen Grid	197
19.2	Sprite Display Size	197
19.3	MOB Corner Screen Locations, Expansion Size 0	198
19.4	MOB Corner Screen Locations, Expansion Size 1	199
19.5	MOB Corner Screen Locations, Expansion Size 2	199
19.6	MOB Corner Screen Locations, Expansion Size 3	200
19.7	Block Numbers and Starting Memory Locations for MOBs	204
21.1	Waveforms Illustration	217
21.2	Waveform Parameter Binary Number	219
21.3	ADSR Envelope Illustration	222
21.4	ADSR Values	223
21.5	Note Duration Table	226
21.6	PLAY Statement Functions	227
21.7	Ring Modulation Voice Number Table	228
21.8	Synchronization Voice Number Table	230
22.1	Joystick Position Values	236

INDEX

Accessing function keys (Figure 4.1), 32
ADSR, 222 224
ADSR envelope, illustration (Figure 21.3), 222
ADSR values (Figure 21.4), 223-224
Allocating sprite memory, 189-191, 207-208
Allocating user-defined character memory, 123-127
ANGL command, statement, 159-161,184
ARC command, statement, 155-159,184
Assigning commands to function keys, 31-34
Assigning commands to function keys, with carriage return, 33-34
AT function, 68-70
@ statement, user-defined characters, 123-127
@ statement, sprites, 191-194
AUTO command, 4-6
Automatic line numbering, *see* AUTO command (4-6)

Background color, *see* LOW COL command (148-149,185)
BCKGNDS command, statement, 108-112
BFLASH command, statement, 115-116
Binary to decimal number conversion, 38
BLOCK command, statement, 179-180
Block numbers, MOB (Figure 19.7), 204-206, *also see* MOB SET statement (194-195, 207-208)
Border
  color, screen, *see* COLOUR command (107-108)
  flashing, *see* BFLASH command (115-116)
Branching statements
  CALL statement, 97-99
  CGOTO statement, 85-88
  EXEC statement, 99
  IF THEN ELSE statement, 80-81 and RCOMP ELSE statement, 81-82
  INKEY statement, 57-59
  LOOP, EXIT IF, END LOOP statement, 83-84
  ON ERROR statement, 21-25
  ON KEY statement, 59-61 and RESUME, 61-62
  REPEAT UNTIL statement, 82-83
  RESET statement, 89-93

CALL statement, 97-99
Calling a closed procedure, 99
Calling an open-ended procedure, 97-99
Cancelling automatic line numbering, 5-6
CENTRE command, statement, 46-48
Centering characters, *see* CENTRE command (46-48)
CGOTO statement, 85-88
Changing automatic line numbering parameters, 6
CHAR command, statement, 145-147
Character background
  color, *see* BCKGNDS command (108-112)
  flashing, *see* FLASH command (112-115)
  reverse field, *see* INV command (55-56)
Character duplication, *see* DUP command (52-54)
Character input, *see* ON KEY statement (59-61)
Character sets, switching between, 54-55
Character string
  expanding, *see* INSERT statement (48-49)
  location within main string, *see* PLACE statement (51-52)
  overwriting, *see* INST statement (49-51)
Characters
  on a graphics screen, *see* CHAR command (145-147), TEXT command (142-145)
  print color, *see* FCOL command (118-119), FILL command (116-118, 126)
  screen placement, *see* AT function (68-70)
  user-defined, 122-127
CHECK statement, 202
CIRCLE command, statement, 152-155, 184
Closed procedure, 96-97, 99-103
CMOB statement, 195-196
COLD command, 12-13
Collision detection, sprites, *see* DETECT and CHECK statements (201-202)
Color
  filling an enclosed shape, *see* PAINT command (165-167, 185), BLOCK command (167-168)
  key press table (Figure 13.1), 108
  multicolor sprites, *see* CMOB statement (195-196)
  numbers (Figure 13.1), 108
  plotting background color in HIRES, *see* LOW COL command (148-149, 185)
  plotting color - see HIRES command (137), MULTI command (181-183)
  plotting color changes - see LOW COL command (148-149, 185), HI COL command (149-150)
COLOUR command, statement, 107-108
Control characters, FETCH, 63
COPY statement, 175-176
CSET command, statement, 54-55
CSET 2 command, statement, 174-175

Data pointers, resetting, 89-93
DELAY command, 11-12
DESIGN statement, 189-191,207-208
DESIGN 2 statement, 123-127
Design grid
  sprites, 191-194
  user-defined characters, 123-127
Design instructions for DRAW command, (Figure 16.1), 169
DETECT statement, 201-202
DIR command, 41-42

DISAPA: statement, 26-27
DISK command, 40-41
Disk directory listing, see DIR command (41-42)
DISPLAY command, 34-35
Display on video screen
  DRAW design, see ROT command (168-170)
  sprite, see MMOB statement (196-201)
DIV function, 36
DOWN command, statement, 129-130
DRAW command design instructions (Figure 16.1), 169
DRAW command, statement, 168-170
DUMP command, statement, 18-19, 20-21
DUP command, statement, 52-54
Duplication
  characters, see DUP command (52-54)
  screen portion, see MOVE command (71-74)

Ellipse, plotting, 153-154
END PROC statement, 99
ENVELOPE, ADSR illustration (Figure 21.3), 222
ENVELOPE statement, 222-224
ERRLN function, 21-25
ERRN function, 21-25
Errors trapped by ON ERROR (Figure 2.2), 21-22
EXEC statement, 99
EXOR function, 38-39

FCHR command, statement, 119
FCOL command, statement, 118-119
FETCH control characters (Figure 8.2), 63
FETCH statement, 62-67
FILL command, statement, 116-117, 126
FIND command, statement, 19
FLASH command, statement, 112-115
FLASH, turning off, see OFF command (113)
Formatting
  characters, see CENTRE command (46-48), AT function (68-70)
  numbers, see USE command (70-71)
FRAC function, 37
Function key
  access for KEY command (Figure 4.1), 32
  assignment, see KEY command (31-34)
  assignment display, see DISPLAY command (34-35)
  input, see INKEY statement (57-59)

GLOBAL statement, 93-95
Graphic
  modes, 133-134
  screen, hardcopy, see COPY statement (175-176)
  screen, high resolution, 134-135
  screen, multicolor, 177-178
  screen recall, see CSET 2 (174-175)
GRAPHICS function, 210

Hexidecimal to decimal number conversion, 38
HI COL command, statement, 149-150
Highlighting Simons' BASIC keywords in listings, see OPTION command (74-75)
High-resolution
  graphics, 134-135
  plot types (Figure 15.1), 138
  screen layout (Figure S5.1), 135
HIRES command, statement, 137
HRDCPY statement, 75-76

IF THEN ELSE statement, 80-81
Initializing
  high-resolution screen, see HIRES command (137)
  multicolor screen, see MULTI command (181-183)
  sprite, see MOB SET statement (194-195, 207-208)
INKEY statement, 57-59
INSERT statement, 48-49
INST statement, 49-51
INV command, statement, 55-56

JOY function, 234-236
Joystick position values (Figure 22.1), 236
Joystick read function, see JOY function (234-236)

KEY command, 31-34
KEY command, with cariage return, 33-34
Key conventions for program entry, 3, 44-45, 106, 188, 213

LEFT command, statement, 129
Light pen read functions, see PENX and PENY functions (238)
LINE command, statement, 150-152
Line numbering
  automatic, see AUTO command (4-6)
  renumbering, see RENUMBER command (6-9)
Listing, disk directory, see DIR command (41-42)
Listings, programs
  highlighting keywords, see OPTION command (74-75)
  paging, see PAGE command (9-11)
  printing speed, see DELAY command (11-12)
Loading a low-resolution screen, see SCRLD command (121)
LOCAL statement, 93-95
LOOP, EXIT IF, END LOOP statement, 83-84
Loops, see Branching statements
LOW COL command, statement, 148-149, 185
Low-resolution screen printing, see HRDCPY (75-76)

Math functions
DIV function, 36
FRAC function, 37
MOD function, 37
MEM statement, 122
Memory allocation
sprites, see DESIGN statement (189-191, 207-208) and MOB SET statement (194-195, 207-208)
user-defined characters, see DESIGN 2 statement (123-127)
MERGE command, 103-104
MMOB statement, 196-201
MOB, see Sprite (MOB)
MOB OFF statement, 203-204
MOB SET statement, 194-195, 207-208
MOD function, 37
MOVE command, statement, 71-74
MULTI command, statement, 181-183
Multicolor
plot types (Figure 18.2), 181
screen configuration, 177-178
screen grid (Figure S6.1), 180
MUSIC statement, 224-226

Naming a procedure, see PROC statement (97)
NO ERROR statement, 21-25
Note Duration Table (Figure 21.5), 226
NRM command, statement, 164
Numeric aids, 36-39
Numeric conversion functions
binary to decimal, 38
exclusive-OR, 38-39
hexidecimal to decimal, 38

OFF command, statement, 113
OLD command, 12
ON ERROR statement, 21-25
ON KEY return statement, see RESUME statement (61-62)
ON KEY statement, 59-61
ON KEY valid characters (Figure 8.1), 60
Open-ended procedure, 96-99
OPTION command, 74-75

Paddle read function, see POT function (237)
PAGE command, 9-11
PAINT command, statement, 165-167, 185
PAUSE command, statement, 20-21
PENX function, 238
PENY function, 238
PLACE statement, 51-52
PLAY functions (Figure 21.6), 227
PLAY statement, 226-228
Plot colors, see HIRES (137), LOW COL (148-149, 185), HI COL (149-150), and MULTI (181-183) commands
PLOT command, statement, 161-162
Plot types
high-resolution (Figure 15.1), 138
multicolor (Figure 18.2), 181
Plotting a square
high-resolution screen, 140-142
multicolor screen, 183-184
Plotting an ellipse, 153-154
Plotting commands, see specific commands: REC (137-142), LINE (150-152), CIRCLE (152-155, 184), ARC (155-159, 184), ANGL (159-161, 184), PLOT (161-162), TEST (162-164)
POT function, 237
PRINT function, see AT function (68-70)
Printing text on a graphics screen, see TEXT command (142-145), CHAR command (145-147)
PROC statement, 97
Procedures, 96-103
Program Listings
highlighting keywords, see OPTION command (74-75)
paging, see PAGE command (9-11)
printing speed, see DELAY command (11-12)
Pulse/square waveform, 220-221
Pulse waveforms, 220-221

Read functions
joystick, see JOY function (234-236)
light pen, see PENX and PENY functions (238)
paddle, see POT (237)
RCOMP ELSE statement, 81-82
REC command, statement, 137-142
Recalling a graphic screen, see CSET 2 (174-175)
Rectangle, plotting, see REC command (137-142)
Redisplaying a saved screen, see SCRLD (121)
RENUMBER command, 6-9
REPEAT UNTIL command, statement, 82-83
RESET statement, 89-93
Resetting data pointers, see RESET statement (89-93)
RESUME statement, 61-62
RETRACE command, 17-18
Reusing variables, see LOCAL and GLOBAL statements (93-95)
Reversing character background, see INV command (55-56)
RIGHT command, statement, 127-129
Ring modulation, 228-229
Ring modulation voice number table (Figure 21.7), 228
RLOCMOB statement, 201
ROT command, statement, 170-173
Rotation
DRAW design, see ROT command (170-173)
instructions (Figure 16.2), 170

Screen
border color, see COLOUR command (107-108)
border, flashing, see BFLASH (115-116)
color, see COLOUR command (107-108)
duplicating a portion, see MOVE command (71-74)
lines, paging by, see PAGE command (9-11)
load, low-resolution, see SCRLD statement (121)
printing, high-resolution, see COPY statement (175-176)
printing, low-resolution, see HRDCPY statement (75-76)
save, low-resolution, see SCRSV (120-121)
scrolling, see RIGHT, LEFT, UP, and DOWN commands (127-132)
Screen grids
low-resolution and high-resolution (Figure S5.1), 135
low-resolution, high-resolution, and multicolor (Figure S6.1), 178
MOB, corner locations, expansion size 0 (Figure 19.3), 198
MOB, corner locations, expansion size 1 (Figure 19.4), 199
MOB, corner locations, expansion size 2 (Figure 19.5), 199
MOB, corner locations, expansion size 3 (Figure 19.6), 200
multicolor (Figure 18.1), 180
sprite screen grid (Figure 19.1), 197
SCRLD command, 121
Scrolling techniques, 127-132
SCRSV statement, 120-121
SECURE command, 27-29
SOUND function, 220-221
Sprite (MOB)
block numbers and memory locations (Figure 19.7), 204-206
collision detection, see DETECT and CHECK statements (201-202)
Sprite (MOB)—continued
color, see @ statement (191-194), CMOB statement (195-196)
design, see @ statement (191-194)
display, see MMOB statement (196-201)
display size (Figure 19.2), 197
initializing, see MOB SET statement (194-195, 207-208)
memory allocation, see DESIGN statement (189-191, 207-208)
movement, see MMOB statement (196-201), RELOCMOB statement (201)
removal, see MOB OFF statement (203-204)
Square
plotted on high-resolution screen, 140-142
plotted on multicolor screen, 183-184
Structured programming, 96-103
Synchronization, 229-231
Synchronization voice number table (Figure 21.8), 230

TEST command, statement, 162-164
TEXT command, statement, 142-145
TEXT height and spacing values (Figure 15.3), 143
TRACE command, statement, 14-18, 20-21, 29
TRACE, turning off, 16

UP command, statement, 130
USE command, statement, 70-71
User-defined characters, 122-127

Variables, reusing, see LOCAL and GLOBAL statements (93-95)
VOL statement, 215-217

WAVE statement, 218-220
Waveform parameter binary number function table (Figure 21.2), 219
Waveforms, 217-218
Waveforms, illustration (Figure 21.1), 217

Commodore's Handbook of Simons' BASIC

Jane Reh

"Thorough"... "Very accurate"... "Easy to read"... "The organization is excellent"... "Progresses nicely from the simple to the complex"... "An excellent resource"

These are just a few of the prepublication reviewers' comments made about Jane Reh's Commodore's Handbook of Simons' BASIC—the first and final word about mastering Simons' BASIC. With a foreword written by David Simons himself, this is a complete reference that carefully details the 144 commands in Simons' BASIC while offering very practical dos and don'ts to make your programming experience a smooth and easy one. For business or for pleasure, you'll be given a hands-on tour through this powerful extension of the Commodore BASIC language with helpful pictorial aids, "illustrated commands," charts, graphs, and screen grids to help you every step of the way. The book also includes an extensive index and Simons' BASIC command quick reference listing.

Don't settle for ordinary BASIC programming when you can enjoy the advantages of Simons' BASIC.

CONTENTS

Foreword by David Simons • Preface • Section I—Programming Aids We All Need • Statement Numbers and Program Listings • Debugging Aids • Program Security Aids • Assigning Commands to the Function Keys • Numeric Aids • Disk Handling • Section II—Added Input and Output Options • Character String Input Options • Input Validation • Output Options • Section III—More Programming Power • Condition Testing and Controlling Loops • Data Handling • Structure Programming • Section IV—Low-Resolution Color and Graphics • Color and Characters • User-Defined Graphics • Section V—High-Resolution Graphics • Plotting • Coloring and Drawing in Hires • Graphics Output • Section VI-Multicolor Graphics • Multicolor Plotting • Section VII—Sprite Graphics • Creating and Using Sprites • Sprites on a High-Resolution or Multicolor Screen • Section VIII—Simons' Sound • Sound Programming • Leaving the Keyboard • Joystick, Paddle, and Light Pen READ Functions • Appendix—Simons' BASIC Command Reference Listing • Index

0 21898 41931

ISBN 0-89303-419-3